(2)

Systematic Detection of Parallelism in Ordinary Programs

César Augusto Quiroz González

Technical Report 351
May 1991

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

Technical Report 351

Systematic Detection of Parallelism in Ordinary Programs

César Augusto Quiroz González
Department of Computer Science
University of Rochester
Rochester, NY 14627

quiroz@cs.rochester.edu

This TR contains a dissertation submitted in partial fulfillment of the requirements for the degree Doctor of Philosophy. I discuss here a general model for compilers that take imperative code written for sequential machines (*ordinary code*) and detect the parallelism in that code that is compatible with the semantics of the underlying programming language. This model is based on the idea of separating the concerns of *parallelism detection* and *parallelism exploitation.*

This separation is made possible by having the detection component provide an explicit representation of the parallelism available in the original code. This explicitly parallel representation is based on a formalization of the notion of permissible execution sequences for a given mass of code. The model discussed here prescribes the structure of the parallelism detector. This structure depends (1) on recognizing a hierarchical structure on a graph representation of the program, and (2) on separately encoding parallelization conditions and effects.

Opportunities for parallelization can then be discovered by traversing the hierarchical structure from the bottom up. During this traversal, progressively larger parts of the program are compared against the independently encoded conditions, and transformed when the conditions are satisfied. The hierarchy guarantees that the results of transforming a piece of a program propagate in time to affect the possible parallelization of larger pieces. Although some of the algorithms used have exponential worst cases for general graphs, their observed behavior on real flow graphs is no worse than quadratic on the size of the original program.

This investigation was supervised by Prof. Doug Baldwin. The work I report here was supported in part by the ONR/DARPA under Research Contract number N00014–82–K–0193, and also by the National Science Foundation under Grant numbers CCR–8320136 and DMC–8613489. The Government of the United States of America has certain rights in this material.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER TR 351	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Systematic Detection of Parallelism in Ordinary Programs		5. TYPE OF REPORT & PERIOD COVERED technical report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Cesar A. Quiroz Gonzalez		8. CONTRACT OR GRANT NUMBER(s) N00014-82-K-0193
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Dept. University of Rochester Rochester, NY, 14627, USA		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS DARPA 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE May 1991
		13. NUMBER OF PAGES 113 pages
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Arlington, VA 22217		15. SECURITY CLASS. (of this report) unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

parallelization; compilers; parallelism; procedural languages

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

(see over)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT

This dissertation discusses a general model for compilers that take imperative code written for sequential machines (*ordinary code*) and detect the parallelism in that code that is compatible with the semantics of the underlying programming language. This model is based on the idea of separating the concerns of *parallelism detection* and *parallelism exploitation*.

This separation is made possible by having the detection component provide an explicit representation of the parallelism available in the original code. This explicitly parallel representation is based on a formalization of the notion of permissible execution sequences for a given mass of code. The model discussed here prescribes the structure of the parallelism detector. This structure depends on (1) recognizing a hierarchical structure on a graph representation of the program, and (2) separately encoding parallelization conditions and effects.

Opportunities for parallelization can then be discovered by traversing the hierarchical structure from the bottom up. During this traversal, progressively larger parts of the program are compared against the independently encoded conditions, and transformed when the conditions are satisfied. The hierarchy guarantees that the results of transforming a piece of a program propagate in time to affect the possible parallelization of larger pieces. Although some of the algorithms used have exponential worst cases for general graphs, their observed behavior on real flow graphs is no worse than quadratic on the size of the original program.

Curriculum Vitae

César Quiroz was born on 28 December 1957, in Lima, Perú. There he grew up and received his initial schooling. He pursued High School studies at the Colegio Salesiano, where, among many other blessings, he received an enduring love for Mathematics and formal thinking in general.

Late in 1972, his father's job acquired a new instrument: an Olivetti *Programma 101* programmable desk calculator. César got exposed to it, and fell under its spell. He spent part of the next summer exploring the possibilities of that little machine. This document is just another consequence of that exhilarating experience.

He started college in 1974, also in Lima, taking General Studies courses both in the Catholic University and in the National Engineering University. After some reflection (and too many 30 and 40-credit semesters), he had to decide between studying Mathematics (in the Catholic University) or studying Electrical Engineering (in the National Enginering University). Mathematics had won his heart by then, and thus he concentrated on those studies.

The time he spent preparing to study engineering still managed to pay off. In 1976 he attended a sequence of no-credit courses offered by the computer center of the National Engineering University. These courses reopened his then dormant interest in computers, and after that there has been no going back.

So, by 1977 he had both concentrated on his Mathematics degree, and taken a part-time job, this time with the computer center of the Catholic University. Eventually his academic pursuits were rewarded with the Bachelor of Science degree in Mathematics (1981) and the Bachelor of Science degree in Statistics (1983). He also completed the course requirements for the Master of Science degree in Mathematics. Concurrently, he was promoted from a part-time Student Assistant (1977) to part-time Programmer (1978), full-time Systems Programmer (1979), and Coordinator of the Systems Programming team (1980–1983), all of this in the Computer Center of the Catholic University.

He started attending the University of Rochester in 1983, where he received the Master of Science degree in Computer Science (1985). During his stay there he held three Teaching Assistantships, and participated in several research projects. Among those: the Polyhedral Recognition project, led by Dana Ballard (Summer of 1985), and the CONSUL design project, led by Doug Baldwin (Spring of 1986). He has been working under the supervision of Doug Baldwin since then. His final oral examination, in defense of the thesis of this dissertation, was on 23 July 1990.

His research currently focuses on Parallelism in Programming Language implementation.

Acknowledgments

I have been immensely fortunate never to walk this road alone. A merciful Providence made sure there was always someone there when I needed help and company. Each line of this work is indebted to the acts of kindness of many people over many years. My sincerest apologies to those I can't mention here: their number is now beyond my capacity to recall in this document. I will try to bring forth at least the major sources of help and inspiration that have made this dissertation possible.

First of all, I want to thank my advisor, Prof. Doug Baldwin. From him I have learned first hand what is involved in doing Science. His support and encouragement were always there when I needed them, especially when I had more trouble making progress. For his generosity, patience, and example in the ways of our profession, my eternal gratitude.

I have also received encouragement and suggestions from my committee: Profs. Jerry Feldman, Robert Fowler, Charles Merriam, and Michael Scott. I also thank Prof. Dana Ballard, for his friendship and for a summer of showing me how far Hough transforms could go (and how much fun making them go that far could be).

Many of the students of the Department have also acquired title to my gratitude. First, my class: Tom Blenko, Paul Chou, Paul Cooper, Kwaku Darkwah, Mark Fanty, Elise Hill, Doug Ierardi, and Jay Weber. CSC400, the first two years of classes, and the Qualifiers, became bearable largely due to their friendship.

Also my thanks to students from other classes (and having been in Rochester for more than seven years, I have met people from *lots* of classes). Alan Cox, Lawrence Crowl, Rick Floyd, Stuart Friedberg, David Sher, and Ken Yap provided countless hours of intellectual stimulus, and always reliable friendship. (Also, numerous rides.) David Coombs, Keri Jackson, Brad Miller, Brian Marsh, Lata Narayanan, and Neil Smithline were always very close to me, especially in difficult times. This list could easily be longer, and new names are added every day. Thanks.

My thanks also to the administrative staff of the Department, without whose help I would have perished a thousand times in the arcane game of University procedure. Jill Orioli Forster, Peggy Frantz, Peggy Meeker, Suzanne Bell, Pat Armstrong: a million thanks. And also to the technical staff, indefatigable wizards and friends. My special gratitude to Tim Becker, Liudvikas Bukys, and Jim Roche. Also to Ray Frank, whose passion for chess I share.

My thanks also to Myra VanInwegen, for more than three years of love and warm company, for having been for me a family when I was so far away from my natural family, for her special backrubs, for warm cinnamon buns and the Sunday New York Times. Her affection kept me going through some difficult times in the course of this research. I wish her much success with her own Ph.D. work, and in whatever path she chooses thereafter.

Many other people contributed significantly to making my tenure in Rochester the pleasurable experience it has been. I want to mention especially the personnel of the University Health Service, and of the Strong Memorial Hospital, for their help in keeping me healthy and in returning me to health several times. Also College Town Pizzeria, for more than just warm food delivered at insane hours. To all of them, thanks.

My gratitude goes also to Richard Stallman and his Free Software Foundation, for the pleasure of using GNU Emacs and the convenience afforded to my research by having access to the GNU C compiler. I acknowledge also the use of Taiichi Yuasa and Masami Hagiya's implementation of Common Lisp, KCl, and also of Bill Schelter's improvement thereon, AKCL. I can only hope that this tradition of sharing good code will grow stronger.

My work in Rochester was supported in part by the ONR/DARPA Research Contract number N00014-82-K-0193. This material is based upon work supported by the National Science Foundation under Grant numbers CCR-8320136 and DMC-8613489. The Government of the United States of America has certain rights in this material.

This dissertation is dedicated to my parents, with all my love *¡Gracias!*

Abstract

This dissertation discusses a general model for compilers that take imperative code written for sequential machines (*ordinary code*) and detect in that code the parallelism that is compatible with the semantics of the underlying programming language. This model is based on the idea of separating the concerns of *parallelism detection* and *parallelism exploitation*.

This separation is made possible by having the detection component provide an explicit representation of the parallelism available in the original code. This explicitly parallel representation is based on a mathematical formalization of the notion of permissible execution sequences for a given mass of code.

Having made that separation, one can discuss an organization for the parallelism detection component. This organization depends (1) on recognizing a hierarchical structure on a graph representation of the program, and (2) on separately encoding parallelization conditions and effects.

Opportunities for parallelization can then be discovered by traversing the hierarchical structure from the bottom up. During this traversal, progressively larger parts of the program are compared against the independently encoded conditions, and transformed when the conditions for a parallelizing transformation are satisfied.

Table of Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1. The Problem

Compilers for sequential imperative code are understood in terms of a well-accepted model: a lexical analyzer tokenizes the input text, a syntactic analyzer parses the tokenized input, a semantic analyzer validates non-context-free attributes of the parsed input, etc. (For examples, see [ASU86, fig. 1.10] and [WaG83, §1.4].) Even when a compiler's code does not clearly exhibit this division of labor, this model helps to understand it, or to compare it with similar compilers.

Thus the standard model is useful in two senses. In the *descriptive* sense, it describes the behavior of existing compilers. In the *prescriptive* sense, it lays out the structure of compilers yet to be written, thereby giving their builders a mental map of what needs to be done.

In this context, let a *parallelizer* be a compiler that converts ordinary code for execution on parallel machines. By *ordinary code* I mean code written in a sequential imperative language, without concern for parallelism. There already exist many successful parallelizers. However, they have all been conceived and implemented as special cases, carefully tuned to specific combinations of input language and target architecture. Although they are impressive accomplishments of the art of compiler design, there is but the vaguest impression of a generalizable design in them. Moreover, there isn't a consensus about a model for them, at least not to the successful extent of the standard model for sequential compilers.

The goal of this dissertation is to provide a model for parallelizing compilers, and to show that the model's descriptive and prescriptive powers are similar to those of the standard model.

1.2. A Model

The model I present here is based on the separation of two concerns: *detection* of parallelism and *exploitation* of parallelism. When designing a new parallelizer (or trying to understand an existing one), my model recommends isolating the task of discovering what parts of the program can be done in parallel (regardless of whether doing so buys us something) from the task of selecting a particular parallel execution. The first task depends only on the semantics of the given programming language; the second depends mainly on the details of the machine that will run the object code.[1]

[1] This separation has consequences elsewhere in parallel programming. For instance, Crowl [Cro91] argues for its importance in achieving architectural adaptability of (explicitly) parallel programs.

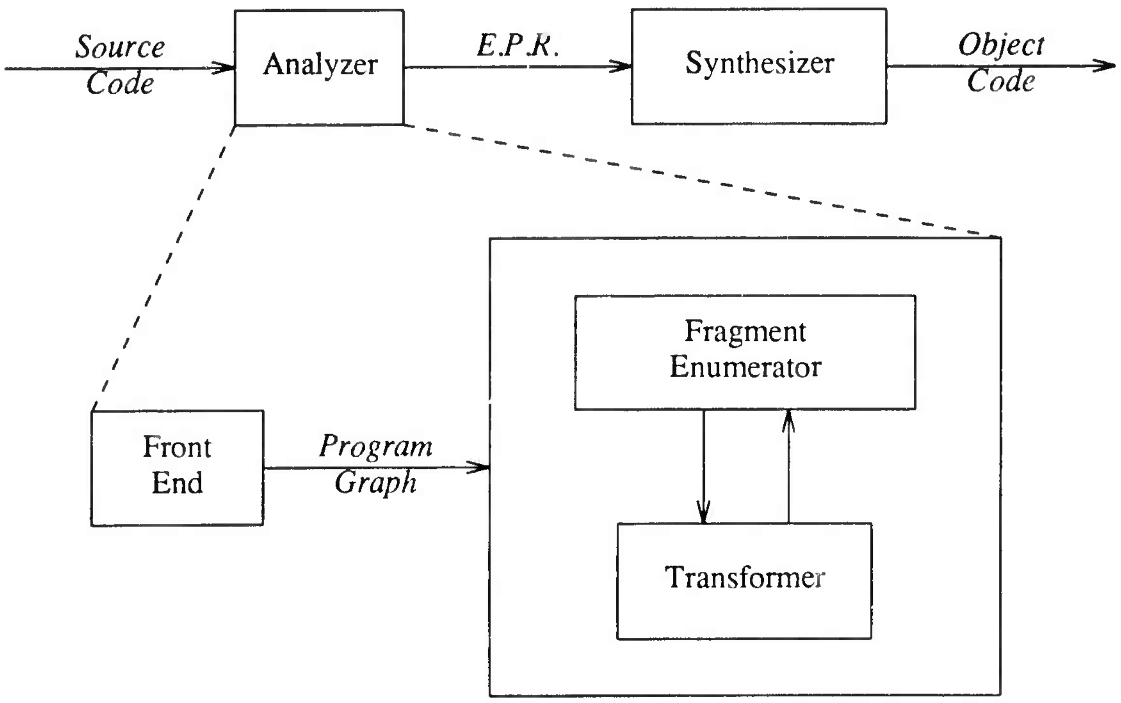


Figure 1.1. A Parallelizer

Figure 1.1 represents this idea. The *Analyzer* is in charge of detecting parallelism, while the *Synthesizer* is in charge of constructing a profitable parallel version of the originally sequential program.

The communication between these two components is a critical aspect in this work. I will discuss later why the separation of concerns requires that the flow from the analyzer to the synthesizer should consist of an *E*xplicitly *P*arallel *R*epresentation of the program.

This dissertation is concerned mainly with the Analyzer (the parallelism detection component). In this model, the source program is first validated and transformed by a classic front end. The output of that first part of the compiler is a graph representation of the program (which I will call the *program graph*). I will assume here that the source program gets reduced to a control flow graph at some granularity (statements or basic blocks are examples). This is the usual representation, see for instance [ASU86, §9.4 and §10.1]. I will call each node of that graph a *granule*, a generic term that avoids pre-judging the actual resolution of the analysis.

In addition, I will assume that data flow information at the granule level is either available, or conveniently computable from the output of the front end. The front end computes this information (control and data flow) on the assumption that the program will be executed sequentially. This assumption defines a partial order on the sequence of execution of the various parts of the program. The purpose of the parallelism detector can be described as finding a relaxation of this partial order.

The first stage of the analyzer proper is the *Fragment Enumerator*. This component will select pieces of the program (I'll call them generically *fragments*) to be considered for parallelization. Fragments formalize a bottom-up strategy for the analysis. Therefore, I need to define fragments in such a way that some containment relation holds between them. Some basic properties are shared by all useful definitions of fragments:

(1) Some minimal subparts of the program (perhaps individual granules) are fragments

(2) The whole program is a fragment, and

(3) Some intermediately sized configurations are fragments too.

Every choice of fragment definition produces a hierarchy of parts of a program, of progressively increasing size.

However, not all hierarchies are equally useful. It is possible to construct hierarchies whose components don't abstract useful properties of the program, or that abstract at a grain too fine or too coarse for practical application. Choosing a suitable definition of fragment (and hence, of fragment hierarchy) will thus be critical to the success of a parallelism detector.

If we further restrict our attention to the definitions that yield practical hierarchies, the choice between granularities permits us to control the tradeoff between thoroughness and cost of the analysis.

Parallelization in this scheme is driven by the fragment hierarchy. Fragments are presented in turn to the *transformer*, an agent in charge of exposing the parallelism in a fragment. The discovery of this parallelism occurs by subjecting the fragment to a series of tests, which involve conditions on the shape of the fragment, its position in the fragment hierarchy, its associated data flow information,[2] and the results of previous parallelizations.

When a fragment is being considered, I would like to take advantage of the results of having previously considered its sub-fragments. So, the fragment enumerator must guarantee the following condition: *If* f *is a fragment of a program graph, then* f *is eventually presented to the transformer. If* f *and* g *are fragments of a*

[2] I assume here that fragment-level flow information can be computed from the basic-block-level flow information provided by the front end. (This is another constraint on the definition of a fragment.) As fragments contain blocks, well-known technology to propagate flow information bottom-up applies here.

program graph, and f *is a proper subgraph of* g, *then* g *is last presented to the transformer after the last presentation of* f. This condition guarantees a form of orderly progress in the enumeration, as it roughly states that large eventually follows small. I will call this condition *eventual enumeration ordering*. (Notice that a fragment can be presented for parallelization more than once.)

The parallelism (or its lack) exposed by the transformer is encoded in an *execution set*. An execution set for a fragment is a set of permutations of parts of the fragment, that contains only permutations whose sequential execution honors the data and control dependences in that fragment. Exposing parallelism in the fragment reduces to finding larger execution sets (Chapter 3).

In order to expose these larger execution sets, the transformer may need to modify the fragment (I call this a *reorganization*) or add some code to the rest of the program (and this I call a *compensation*; these two terms are standard). The transformer does not have hard-coded information about the needed tests and transformations: the required tests, reorganizations, compensations, and calculations of execution sets are encoded in another structure, the *parallelizations catalog*.

Eventually, the enumerator will present for possible transformation the fragment corresponding to the total program, for the last time. After that consideration, the resulting execution set is the final output of the analyzer.

Figure 1.2 shows the flow of information within the Analyzer. To a first approximation, the Analyzer is a pipeline composed of three stages. The pipeline model is not entirely adequate, because each stage keeps state that is visible to the others, but will serve us for an introduction. So, let's follow the left-to-right pointing arrows in Figure 1.2.

The first stage is the Fragment Enumerator. This consumes (notionally) the program graph and produces fragments. Those fragments are given execution sets by the

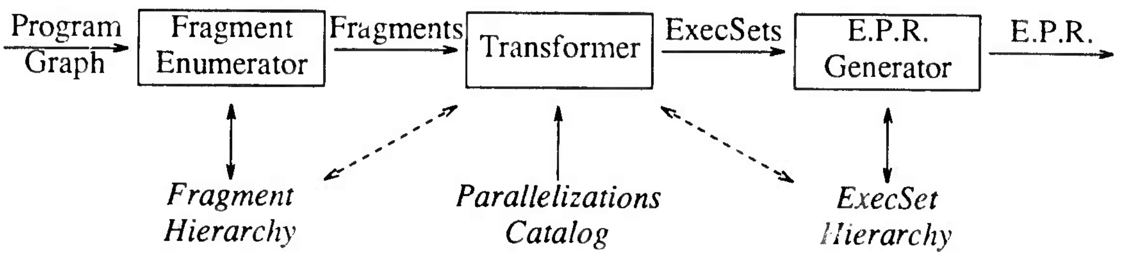


Figure 1.2. Information Flow in the Analyzer

second stage, the Transformer. Finally, the last stage in the figure, the *E.P.R. generator*, outputs the final execution sets, in a notation useful to the *synthesizer* component. That notation is what I call an *Explicitly Parallel Representation*. (Of course, the E.P.R. can just coincide with whatever notation we use for execution sets.)

The stages, as said above, keep state too. The solid double-headed arrows represent creation and modification of two major data structures: the Fragment and ExecSet hierarchies. These are effectively shared by all the stages. In addition, a static structure is queried by the Transformer: the Parallelizations Catalog, which encodes the transformations that have been deemed worth trying.

Finally, the dashed double-headed arrows represent a more serious potential departure from the pipeline model. The transformer could (by virtue of reorganization or compensation) modify the flow structure of the program to the extent of forcing the calculation of a new Fragment Hierarchy. As the possible new fragments would require execution sets too, the ExecSet hierarchy would be naturally modified too.

1.3. The Thesis

The thesis defended by this dissertation is composed of two claims.

The first claim is that the detection of parallelism in ordinary programs can be performed independently of the architecture of the machine that will run them, thereby separating this detection from the problem of selecting useful parallelism, and that such separation leads to a very general model of how parallelizers work.

I will support this claim by showing a model of parallelization that realizes this separation.

The second claim is that the generality obtained by this model is not a serious impediment to effective implementation. It is at least conceivable that the desired separation would sever important links between the two parts posited, and that recovering that connection could be costly. Or, it could be that the generality of the model depends on algorithms that are inherently too expensive. In either case, compilers designed around my model could well be too slow to compete practically with others crafted in less systematic, but adequate, ways. I claim that no such crippling inefficiencies arise from my model.

I will support this claim by observing the behavior of a proof-of-concept implementation of a parallelism detector for a subset of C [KeR78], named SPOIL.[3]

About the subset of C compiled by SPOIL, suffice it to say that it is a simple imperative language, with only first order features (various scalar types, some structured types like records and arrays, functions, some form of looping). The target

[3] Systematic Parallelization Of Imperative Languages. Note also the play on the fact that *to compile* derives from a French word that used to mean *to plunder*.

machine is left unspecified, as it is part of the claim that this is not a matter with which the parallelism detector should concern itself.

The rest of this dissertation discusses the parallelism detection component of a parallelizer for a generic imperative language, and describes SPOIL as an application of this model for the specific case of C.

It is important to pay attention to three levels of abstraction represented in SPOIL. At the highest level, SPOIL is a representative of the compilers that follow my model. Thus, it separates detection from exploitation, is driven by a hierarchy of program parts, separates the compiler's logic from the specific transformations applied, and represents parallelism explicitly. At this level, SPOIL can be judged by whether (or how well) it adheres to the model. Conversely, SPOIL helps us judge the usefulness of the basic tenets of the model, by providing existence proof of the model's applicability.

Among those compilers that follow the model, SPOIL is distinguished (second level) by specific definitions of granule, fragment, and aspect (these will be introduced later) and by a specific catalog of parallelizing transformations. At this level, SPOIL can be judged by how well it succeeds in using those concepts to parallelize the specific subset of C that it compiles. Conversely, having observed its success in those terms, SPOIL helps establish the claim that languages in such class can be treated by my model. The critical feature of C used at this level is its amenability to effective data and control flow analysis. That the class of languages for which this level applies is well-defined can be seen by considering that there are imperative languages that *don't* fit this characterization (say, the higher-order languages studied in [Ney88]).

Finally, at the third (and lowest) level, SPOIL is a particular Common Lisp program running under the UNIX operating system and compiling specific C codes. At this level one could judge the implementation characteristics of SPOIL, like its efficiency and effectiveness on a test suite. That these parameters end up being acceptable (see Chapter 6) supports the claim that constructing this class of parallelizer is not only *possible*, but also can be *useful.*

I distinguish these three levels in order to separate the effects due to the model itself from effects due to specific model-related decisions (like granularity or choice of transformations), and from accidental effects of the implementation. The interplay of these three levels is easier to understand by considering an example.

First, my model (the higher level) recommends that we represent the program as a hierarchy of parts, and that our compiler go through the hierarchy in the order of increasing size. When dealing with some specific language, we are to design a representation that makes this hierarchical traversal possible. It also has to be possible to identify promising parts of the program, possibly by matching them against known shapes.

At the second level, we choose a specific representation for a language of interest. For instance, we bind the notion of granule to, say, basic blocks. We also have to define fragments somehow, perhaps we choose single-entry/single-exit regions of the flow graph. Now we have to decide how to recognize interesting shapes. For the C subset used in this investigation, I used graph parsing with respect to the SSFG grammar of Farrow, Kennedy and Zucconi [FKZ76]. We also choose well-known parallelizations, to show that they can be expressed in the formalism of my model.

At the bottom level we have the decisions that enter the writing of the specific parallelizer. These may include file system formats, mode of interaction with the user, etc. For SPOIL, Chapter 6 provides some pertinent detail.

The second level is especially important for this thesis. The highest level is so general that it is difficult to evaluate its relevance by means of testable claims. The bottom-most level is so crowded with implementation details that it is hard to extract useful generalizations from it (that, indeed, is the characteristic of the previous work that most heavily pressed me to undertake this study). It is at the second level that I can make testable claims of useful generality. I choose a program representation by means of which I expect to encode the conditions under which a parallelization is possible, and claim that such representation indeed shows existentially that the general model is applicable in a practical setting.

1.4. Previous Work

The main motivation for extracting parallelism from ordinary code is that ordinary code abounds. After the advent of problem-oriented programming languages, like FORTRAN, most computer systems presented the programmer with a serial mode of operation, where the concurrency between the various functional units of the computer was hidden. And thus a wealth of code has been written by people blissfully unaware of concurrent execution possibilities.

Essentially all of the popular problem-oriented programming languages have been defined in terms of sequential semantics. Thus, whether the programmer intends this or not, there exists in those languages a well established order of execution. So, the availability of more than one functional unit on a single processor, or of several processors in the same system, is not necessarily a boon to the execution of such programs: it is not immediately clear what tasks to assign to anything but the first such functional unit or processor.

An early attempt at characterizing the conditions under which one can relax the default semantics was reported by Bernstein [Ber66]. His three conditions (discussed later in Section 5.3.4) for independence of two pieces of a program are a foundation for the work that followed. First, they characterize the possible independence of two parts of a program via their memory reference patterns. Interfering patterns prevent parallel execution. Second, the conditions leave open the question of defining what a ``part'' of a program is.

The natural inclination then was to consider the *basic blocks* of the program as the ``parts'' in the conditions. A basic block is a list of instructions that contains no branches, except perhaps as its last instruction, and no jump targets, except into its first instruction (the *leader*). So, once control reaches the leader, all the other instructions are guaranteed execution. Conversely, unless the leader is activated, none of the other instructions is. This all-or-none property makes basic blocks an appealing unit of decomposition of the program.

Basic blocks are clearly larger (in their effect on the store) than individual instructions. However, they are not much larger. Tjaden and Flynn [TjF70] and Foster and Riseman [FoR72] noticed that there isn't much parallelism available inside a typical basic block (mainly because there isn't much going on in a typical basic block). So, it seems one would prefer to look at coarser partitions of the program. On the other hand, we can't ignore whatever parallelism is present at (or under) the basic block level, as it is the base case for the rest of the analysis, and considerably easier than the across-block detection anyway. Therefore, all the parallelizers to date have made an effort to enlarge basic blocks, or to look at structures that are potentially larger, while conserving some of the relative simplicity of intra-block parallelization.

Another aspect in Bernstein's conditions was their expression in terms of the state of the store. Two sequential "parts" will fail to be parallelizable if the effect of the sequence of reads and writes to the store determined by the original sequence cannot be maintained except by enforcing the original sequence. In such case, one says that there are *dependences* between the two parts.

The first successful attempts at practical parallelism detection were based on a careful study of those dependences. Kuck [Ban79, Kuc75] and his students and colleagues at University of Illinois produced *Parafrase*, a FORTRAN compiler which had a vectorization facility. A new generation of this tool, Parafrase-2, is described in [Pol88].

Similar work was pursued elsewhere, especially at Rice University ([AlK82, ACK87, Ken80], and IBM T.J.Watson ([ABC87]). These efforts constitute the recognized mainstream in the literature. Their main common point is the use of FORTRAN as their target language.

Much of the literature of the late 70s was devoted to FORTRAN on vector machines. Recently, C [KeR78, KeR89] has received some attention too. Parafrase-2 is expected to compile both Fortran and C. And Allen and Johnson [AlJ88] describe a C compiler for Ardent's Titan.

The recognition that standard programming languages may make the recognition of parallelism difficult (mainly on account of the problem of resolving aliased references to the store) has induced some workers to consider compilers that interact with a knowledgeable user. The compiler (actually, a programming environment developed around a compiler) would try to relax dependences as far as possible, and submit to the human user the consideration of those dependences that appear promising but couldn't be relaxed automatically. Examples of this approach are [ABC87, SmA88].

The FORTRAN-centered efforts are driven by the desire to speed up scientific code. This means that symbolic computation was neglected at the beginning, as this is not an activity for which FORTRAN is well suited. In recent times, Scheme [Lar89] and Common Lisp [Har86, PHK88] have received some attention.

While iteration is the natural habitat for parallelism in FORTRAN and similar languages, it is not necessarily so for Lisp. So one can expect that the Lisp parallelizers use techniques unusual in the FORTRAN world. Still, there are important common points: One can calculate statically a reasonable approximation to the dependence relationship between parts of the program. The notion of dependence, explored in Chapter 2, underlies the necessary analyses. For the difficulty in establishing accurate control flow information in languages like Scheme, one can refer to [Shi88]. I expect my model to cover the FORTRAN-in-LISP variety of numerical programs as easily as for any other imperative language. In this sense, it should be possible to reproduce Harrison's [Har86] results. Further work is needed to establish this, and to explore the interaction of higher-level imperative features (like mapping) with this model of parallelization. Interprocedural analysis is needed to approach the results reported by Larus [Lar89], this hasn't been tried yet in the context of my model but is a natural direction for extension.

Another dimension of variety is provided by the hardware used to run parallelized code. For VLIW architectures [Fis83], there is J. Ellis's Bulldog compiler [Ell86]. For the WARP [AAG87, KuM84], there is M. Lam's compiler [Lam89]. Finally, for data flow machines, a reference of choice is A. Veen's SUMMER compiler [Vee85].

All these (and other) projects have produced a body of experience on how to build a parallelizer. As mentioned above, one motivation for the work reported here was the absence of a synthesis. There exist many specific parallelizers, but little in the way of a generalization. In terms of the three-level description of these compilers, one can say that most of the literature has been concerned with the lowest and second lowest levels: specific implementation and some language-specific theory.

Recent works have summarized the technical aspects of this experience [Pol88, Wol89]. Reference [Pol88, Figure 1.3] contains a model for Parafrase. That model distinguishes high- and low-level optimizations (those common to a class of architectures vs. those specific to a given machine). The distinction is similar to the one between the analyzer and the synthesizer in my model. What distinguishes my model is (1) the usage of a formally described, explicitly parallel representation as the interface between the high- and low-level parts of the compiler, (2) the treatment of hierarchical compositions of subparts of the program, and (3) the encoding of the transformation conditions and consequences outside the compiler.

1.5. Plan of the Dissertation

Each of the next four chapters has this design: First I discuss the theoretical aspect of the model that supports the first claim of the thesis. Then I consider how this model has been implemented in SPOIL. Then I discuss some related work and the implications of this model.

Chapter 2 describes the input demanded by the parallelism detector. It is essentially a survey of the state of the art in analysis of sequential programs.

Chapter 3 describes the output of the parallelism detector. A mathematical formalization of the notion of permissible execution orders is used to provide a meaning for the transformations that the detector is expected to perform.

Chapter 4 describes the hierarchical structure of program fragments, which provides an organized search space for the parallelism detector. This organization guarantees that the results of parallelizing part of a program may be used when trying to parallelize a larger part. It also provides for a tradeoff between speed of compilation and thoroughness of the parallelization.

Chapter 5 deals with the notion of program transformation that is relevant to this work. By separating the search strategy of the parallelizer from the semantics of the objective function, this model permits another dimension of tradeoff of compilation speed and thoroughness of results.

Chapter 6 summarizes the experience gained through SPOIL. Chapter 7 draws conclusions about the generality and usefulness of this model, argues for the thesis stated above, and proposes future directions of work.

Figure 1.3 shows the correspondence between chapters of this work and parts of the model.

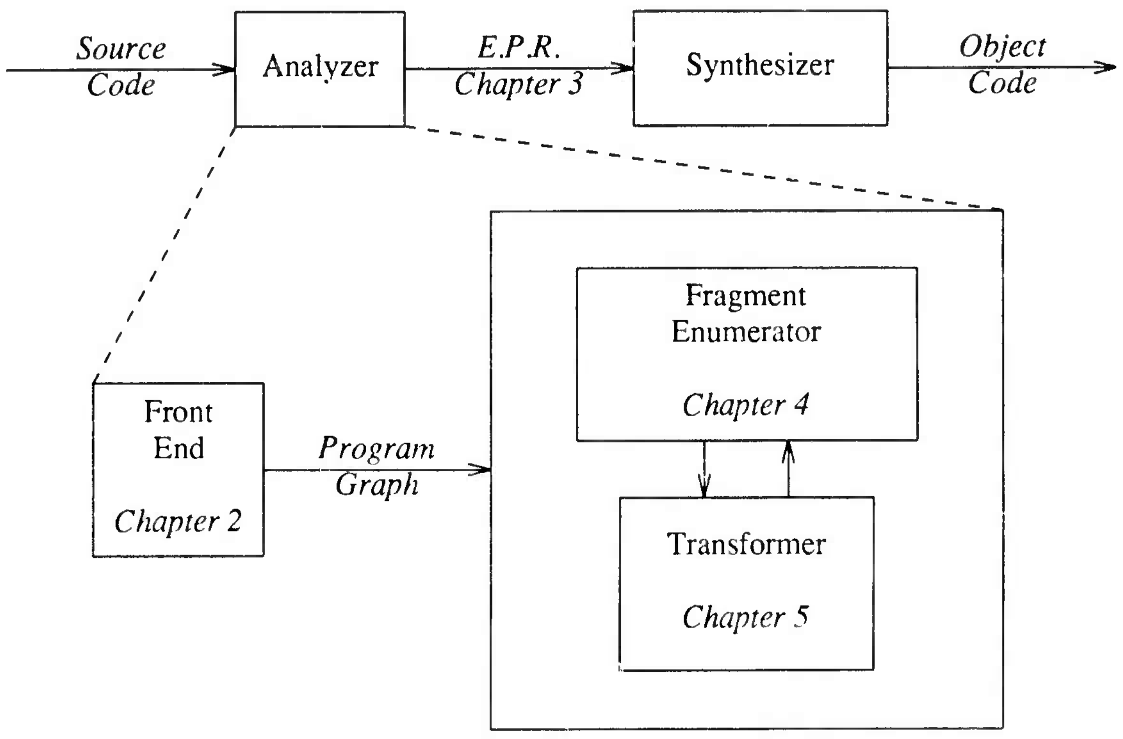


Figure 1.3. Dissertation Map

CHAPTER 2

Input to the Parallelism Detector

The goal of the front end in Figure 1.1 is to validate the syntax and semantics of the input program, and then to provide the parallelizer proper with a convenient representation of the original program. The intention here is to abstract the essential structure of the program from the inessential syntactic details of the source code. The structure that is abstracted consists of control and data relationships between parts of the program. These relationships are used to discover the *dependences* that constrain our ability to execute the program in parallel. In what follows, the assumed representation is a control flow graph with data flow annotations.

2.1. Program Graphs

Consider the program in Figure 2.1. The purpose of that code is to compute the minimum and maximum values of the array **x** (of size **n**), and return them via the reference variables **min** and **max**.

Assuming that no errors are discovered in this program, the front end will produce some internal representation. Figure 2.2 shows an approximation to the three-address code (or the assembly code) that a classic front end could produce from the **minmax** program.

From this representation one can construct another that captures better the way control flows through the program. Figure 2.2 already shows, in comments, the boundaries of the *basic blocks* of **minmax**. A graph whose nodes stand for the basic

```
proc minmax (array x[n]; var min,max);
begin
    min := max := x[0];
    for int i = 1 to n-1 do
        if x[i] < min then min := x[i] end;
        if x[i] > max then max := x[i] end;
    end;
end.
```

Figure 2.1. Minimum and Maximum: Code

```
        # START
        max  := x[0]                  # Basic Block 1
        min  := max
        i    := 1
        $top := n-1
    0:  if i >= $top goto 3           # Basic Block 2
        if x[i] >= min goto 1         # Basic Block 3
        min := x[i]                   # Basic Block 4
    1:  if x[i] <= max goto 2         # Basic Block 5
        max := x[i]                   # Basic Block 6
    2:  i := i+1                      # Basic Block 7
        goto 0
    3:  return                        # STOP
```

Figure 2.2. Minimum and Maximum: Assembly Code

blocks and which has an arc from block B_1 to block B_2 if and only if control can go from B_1 to B_2 is called a *control flow graph* (or, for short, *flow graph*). For more detail, consult [ASU86, §10.4].

Instead of choosing basic blocks as the nodes of flow graphs, one could choose the 3-address statements themselves, or some other unit of program activity. As I would like to separate the design of my Analyzer from the particular choice of control flow granularity, I introduce the notion of *granule*. That is the generic name that I will use in what follows to describe the unit of analysis resolution. (This is, the nodes of the flow graph obtained from the front end.)

In Chapter 3 I develop a formalization for parallel execution of ordinary programs. Note that, in that formalization, granules serve also as units of parallel execution.

The rest of this section is a quick review of the state of the art in control and data flow analysis. Again, [ASU86, Chapter 10] provides a convenient reference.

Figure 2.3 shows a flow graph for **minmax**. This graph captures the important aspects of the way control flows through the code. For instance, we can infer from the graph that control will go from block *BB* 3 to *BB* 4 when the comparison **x[i] < min** is true. So we can say that *BB* 3 is a *predecessor* of *BB* 4 (conversely, that *BB* 4 is a *successor* of *BB* 3). Notice that *BB* 3 has another successor, namely *BB* 5, which is also the successor of *BB* 4. Also, *BB* 3 is preceded by *BB* 2. The information about *BB* 3 and *BB* 4 can be summarized:

$succ(BB\,3) = \{BB\,4,\ BB\,5\}$

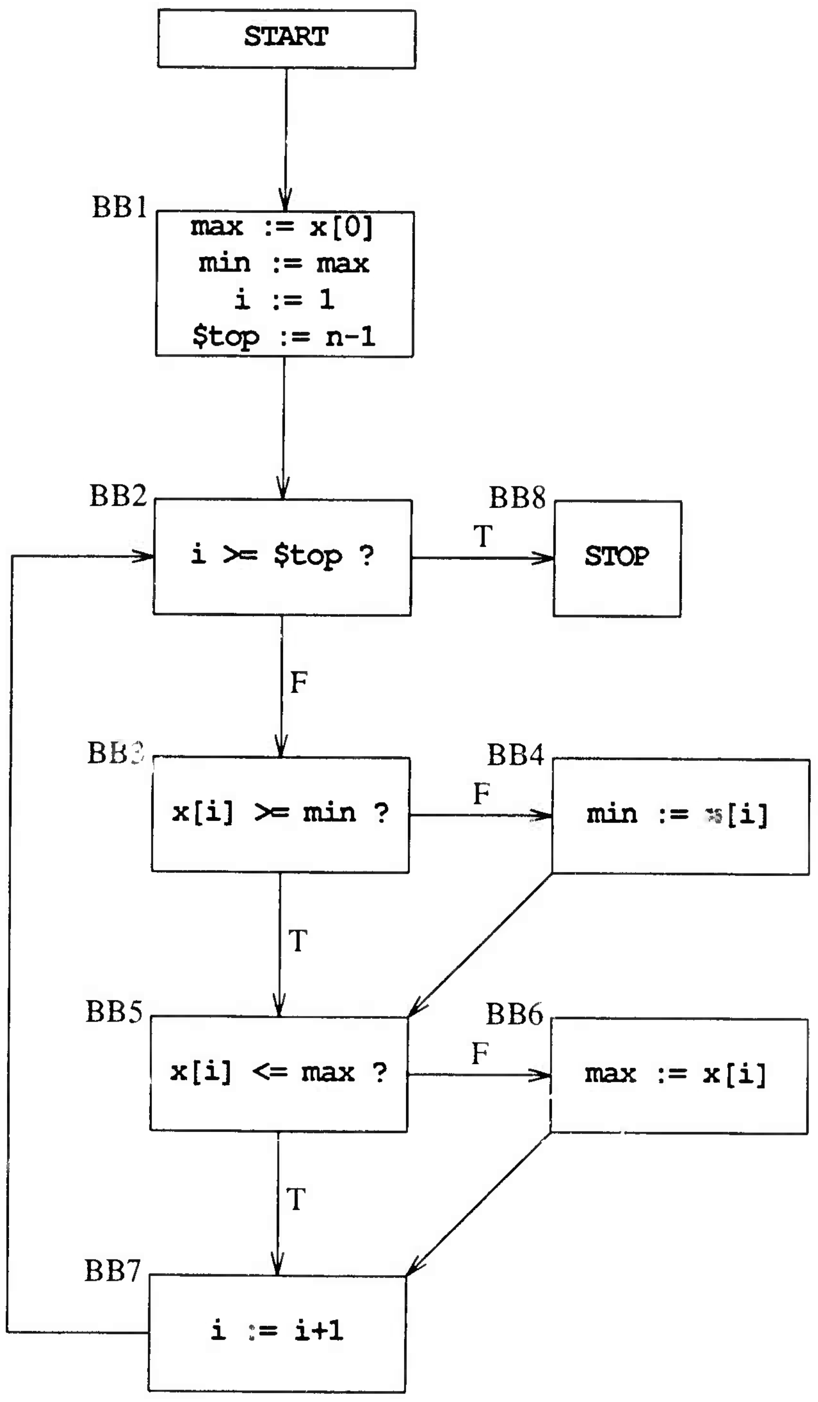


Figure 2.3. A Basic-Blocks Flow Graph for Minmax

$pred(BB\,3) = \{BB\,2\}$

$succ(BB\,4) = \{BB\,5\}$

$pred(BB\,4) = \{BB\,3\}$

Similarly, one can discover that there are paths from block $BB\,2$ to each of the blocks BBx, for $x = 3, 4, 5, 6, 7$[1]. Moreover, all paths from the **START** node to those other blocks pass through $BB\,2$. Consequently, we say that $BB\,2$ *dominates* all those other blocks. These notions of succession and domination are examples of the *control flow relations* that can be discovered from the flow graph. Their discovery is called *control flow analysis*.

Finally, we can use the control flow relations to determine other properties of the program. For instance, the fact that $BB\,2$ dominates $L = \{BBx,\ \text{for}\, x = 3, 4, 5, 6, 7\}$ and that control only leaves L to go to $BB\,2$ ($succ(L) = \{BB\,2\}$) tells us that $\{BB\,2\} \cup L$ is a *loop*, whose *header* is $BB\,2$ and whose *body* is L.

In addition to the control flow information, one can also analyze how data is produced and consumed by the various parts of the program. For instance, we can notice that block $BB\,7$ both *uses* and *defines* a value for **i**. This definition of **i** *kills* the one in $BB\,1$. Because of the loop we mentioned above, the value of **i** that reaches block $BB\,7$ can be either the one set in block $BB\,1$ or the one set in a previous execution of $BB\,7$. If we number each definition by the line number of the three-address statement in which it appears (refer to Figure 2.2), we can describe what we just learned about $BB\,7$ in the following way:

$use(BB\,7) = \{\mathbf{i}\}$

$def(BB\,7) = \{\mathbf{i}\}$

$kill(BB\,7) = \{3,\ 10\}$

$gen(BB\,7) = \{3,\ 10\}$

This information exemplifies the *data flow relations* in the program. For details of the *data flow analysis* procedures, see [ASU86, §10.5].

2.2. The Flow-Attributes Structure

The output of the front end is the collection of control and data flow relations it computes. In the rest of this work it is useful to abstract this information as a function that associates flow information with a piece of a program.

---

[1] And to block $BB\,8$, but here I am interested only in the flow of control from $BB\,2$ into the loop body.

I am going to represent this association by means of a function **Flow** which will take a granule and an *attribute*, and will return whatever control or data flow information is associated with that attribute for that program.[2]

As examples, I will write:

$$Flow(BB\,3,\, succ) = \{BB\,4,\, BB\,5\}$$

$$Flow(BB\,7,\, use) = \{i\}$$

In the examples above, the first arguments have been basic blocks. Later I will generalize this usage to larger units of code.

2.3. Related Work

The information contained in the program graph can, of course, be rearranged in ways different from the ones I postulated in the previous two sections. I notice here a couple of variations that are worth keeping in mind.

First of all, the fact that control flow is represented as a graph means that some control flow relations can be identified by detecting patterns in graphs (more on this later). Of course, data flow relations can also be represented as graphs. This suggests trying to represent the data flow relations *in the same graph* where the control flow information is recorded. See, for examples, [AKP83], or [FeO83], or [FOW87]. I keep an eye open on the possibility of using one of these representations to unify the methods described later. However, there aren't general and efficient graph matching techniques, or at least matching techniques that are efficient for the combined representation. On the other hand, the SSFG parser is well suited to the usual features of control graphs, and it is reasonably cheap, being linear in the size of the graph being considered.

Another important idea is that some blocks compute Boolean values that control the flow in the rest of the program. I have indicated (by means of a question mark after the controlling expression) the conditions computed by those blocks in Figure 2.3. This sort of control dependence information can be computed very effectively independently of the data flow computations (see [CyF88] for an efficient method).

[2] This function is later extended to *fragments* and *aspects*, two concepts briefly touched upon in the introduction, and defined in Chapter 4.

CHAPTER 3

Output from the Parallelism Detector

3.1. Explicit Parallelism

The dependences provided by the front end are an implicit expression of the available parallelism. For instance, in Figure 2.3, we could discover that the part of the program that corresponds to blocks $BB\,3$ and $BB\,4$ can be executed in any order with respect to the part composed of $BB\,5$ and $BB\,6$. (The reason being that the minimum and the maximum can be updated independently.)

However, the implicit information encoded in the dependences cannot, by itself, be used to decouple the parallelism detection from the parallelism exploitation. Indeed, what the dependences actually tell us is *negative* information. If two parts are dependent, then there is no parallelism between them. Classical dependence analysis tells us explicitly where the parallelism isn't, but the synthesizer of Figure 1.1 needs to know where the parallelism is. Otherwise, the synthesizer could not restrict itself to picking between parallel representations; it would have to construct them on demand too.

The counterpart to this observation is that a grossly explicit characterization of the available parallelism is also inadequate. A program with n independent and indivisible parts can be executed in $n!$ different sequences, whose explicit enumeration would be too much data to pass around. (Considering that this example is the best case for parallelization, it would be ironic if it couldn't be handled.) So, the explicit parallelism must be represented in sub-exponential space, if the separation of detection and exploitation is to be practical. I will refer to this property as the *conciseness* of the explicitly parallel representation.

3.2. Execution Sets

The synthesizer is charged w[illegible] producing a program that is both *equivalent* to the original and *parallel*, and for this it will use the explicitly parallel representation that will be presented soon. This section formalizes the notion of safe parallel execution of a sequential program. The view here is static: I will introduce a quantity correlated with the parallelizability of a piece of code. This metric captures nicely the notion of safe parallel execution. I will formalize in this chapter the notion of *making a piece of code more parallel* by relating it to the increase of that metric. On the other hand, I won't explain yet how to go from a less parallel to a more parallel situation (that is the task of Chapter 5).

Given a piece of code, decomposed in several parts, one can entertain the idea of executing those parts in various sequential orders. As a first example, think simply of a piece of code decomposed in individual instructions. One can consider what happens when the instructions are permuted somehow and executed in a sequence other than the textual one.

Of course, most permutations of a program are either nonsense (because illegal) or a different program altogether. For instance, if one were to swap the initialization of a variable with its first use, the use would find an undefined value that would probably cause a failure to continue executing, or would produce a different final value.

However, there are some permutations that leave the meaning of the program untouched. For instance, consider this piece of code:

```
/* i1 */   x = x + 1
/* i2 */   y = 1
```

Because the variables used and modified by one instruction are neither used nor modified by the other, it should be clear that the code below is equivalent:

```
/* i2 */   y = 1
/* i1 */   x = x + 1
```

So, the sequential execution of i_1 followed by i_2 is equivalent to the sequential execution of i_2 followed by i_1. This observation implies that the *parallel* execution of i_1 and i_2 is also equivalent to the original program. Therefore, statements about sequential executions of permutations of a piece of code can be used to derive statements about parallel executions.

This view of parallelism is not new, as it represents properly what happens in many parallel contexts: there is some common clock, or a level of atomic operation, that permits one to interpret parallelism as interleaving of sequential executions. Equivalently, a legitimate execution of a parallel program has the same effects as *some* sequential execution of its parts (this is the classic notion of *serializability*). This is the largely implicit assumption in the early work, as can be seen in [Ber66, Gil58, RaG69(Fig.2)]. Of course, this assumption is generally correct, at least for closely coupled machines.

The advantage of making this assumption is that it enables the use of analysis technology developed for the sequential case. The disadvantage is that it makes it hard to study the case of processors running on different clocks and sharing non-atomic stores.

The connection between parallel executions and interleavings of serial executions makes it appealing to study permutations of sequential code. (Actually, I need to consider permutations *of subsets* of sequential code.) To begin this study, consider a set M and a partial order '$\langle$' on M. Let m be the cardinality of M. A *permutation* π of M is an injective function of the segment of the naturals $\{0, 1, ..., m-1\}$ onto M. As usual, the notation $\mathbf{Z}_m$ will represent $\{0, 1, ..., m-1\}$. We say that π is *compatible* with '$\langle$' if and only if

$$(\forall\, x, y \in M)\, (\forall\, i, j \in \mathbf{Z}_m) \left[(x{=}\pi(i) \wedge y{=}\pi(j) \wedge x \langle y) \Rightarrow i<j \right] \qquad (3.1)$$

Equation 3.1 says that π enumerates the elements of M in a sequence compatible with the order '$\langle$'.

Now consider a program graph. The control flow and data flow relations computed in the front end induce a partial order on the nodes of the program. (I call those nodes the *mass* of the program). That order, as seen by the front end, constrains very rigidly the sequence in which the mass can be executed. Indeed, the front end operates under the assumption that the code will be executed sequentially from beginning to end, modulo the bypasses produced by conditionals and the repetitions produced by loops (this is an equivalent version of the hypothesis of *ordinariness* of the input code).

In terms of order-compatibility, the front end assumes that only one permutation of the granules is compatible with both the control and data flow relations associated with the program graph, once input is provided to the program. In the **minmax** example, the program graph indicates the following order:

$$BB\,1 \langle BB\,2 \langle BB\,3 \langle BB\,4 \langle BB\,5 \langle BB\,6 \langle BB\,7 \qquad (3.2)$$

(where $x \langle y$ is read x *before* y).

If we consider only the body of the loop:

```
 6-         if x[i] >= min goto 1        # Basic Block 3
 7-         min := x[i]                  # Basic Block 4
 8-     1:  if x[i] <= max goto 2        # Basic Block 5
 9-         max := x[i]                  # Basic Block 6
10-     2:  ...
```

we can observe that the following execution sequences are possible (in the sense of respecting the partial order of eq. 3.2):

($BB\,3$, $BB\,4$, $BB\,5$, $BB\,6$)
($BB\,3$, $BB\,4$, $BB\,5$)
($BB\,3$, $BB\,5$, $BB\,6$)
($BB\,3$, $BB\,5$)

Each of the above sequences satisfies the constraints imposed by the order of eq. 3.2. Actually, the first one cannot be realized from any input data, but that cannot be discovered by looking only at the piece of program above. (The invariant **min≤max** has to be established elsewhere.)

We have already noticed that the combination of $BB\,3$ and $BB\,4$ need not be executed before the combination of $BB\,5$ and $BB\,6$ So, in addition to the order (eq. 3.2), we already know that this order is also permissible:

$$BB\,1 \langle BB\,2 \langle BB\,5 \langle BB\,6 \langle BB\,3 \langle BB\,4 \langle BB\,7 \qquad (3.3)$$

Once this is noticed, we are free to consider a larger set of execution sequences that are compatible with the meaning of the original code:

($BB\,3$, $BB\,4$, $BB\,5$, $BB\,6$)
($BB\,3$, $BB\,4$, $BB\,5$)
($BB\,3$, $BB\,5$, $BB\,6$)
($BB\,3$, $BB\,5$)

but also

$(BB\,5, BB\,6, BB\,3, BB\,4)$
$(BB\,5, BB\,3, BB\,4)$
$(BB\,5, BB\,6, BB\,3)$
$(BB\,5, BB\,3)$

The *execution set* of a mass of code with a partial order is the set of the execution sequences of that mass that are compatible with the partial order.

The purpose of the parallelism detector can then be expressed as the construction of the largest (respectively, least restrictive) execution set (respectively, partial order) for the mass of the program. This is the promised connection between a metric (size of the execution set) and the informal notion of ''degree of parallelizability''.

Individual parallelizations can be explained this way: Given an execution set for a program, a *parallelization* is a transformation that yields a larger execution set. (*Transformations* are more precisely described in Chapter 5.) A parallelization is called *safe* if the paths in the new execution set remain compatible with the dependences. The ideal execution set for a program (which may not be discoverable in the absence of perfect information about the program's inputs) typically includes many more paths than are present in the approximations that real compilers can be expected to produce.

Notice the following interesting property: the execution set of a program is composed of sequential enumerations of its mass, yet its size gives us a very strong indication of potential parallelism in that mass. This is so because the permissibility of parallel executions induces many compatible sequential executions. So, a more parallel interpretation of a piece of code necessarily has a larger execution set. Although the converse is not necessarily true, an increase in execution set size at least makes the discovery of parallelism more likely.

Finally, a note on terminology. The sequences shown above (for instance, $(BB\,4, BB\,5, BB\,3)$) do not involve all the mass of the set under consideration. Technically, they are called *selections* or *variations* on that mass, the name permutations being reserved for the case in which all the elements are involved. This distinction is generally immaterial to this discussion, where for convenience I use the term *permutation* as shorthand for *permutation of a subset*, which in Combinatorics would receive one of the names mentioned above.

3.3. Execution Set Notation

Execution sets admit a concise representation. I show here a notation that expresses execution sets by means of certain symbolic expressions (S-expressions). To each such *execution set expression* corresponds exactly one execution set; such expressions encode exponential numbers of execution sequences in sub-exponential space.

The simplest execution set corresponds to individual masses of code, for which I just write **(execset** *mass***)**.

The execution set corresponding to scheduling several others in sequential order will be represented by **(series** $execset_1 \cdots execset_n$**)**. By scheduling in sequential order I mean the complete execution of all the code of $execset_i$ before the execution of any of the code of $execset_j$, for any j greater than i.

The execution set corresponding to scheduling several others in parallel will be represented by **(parallel** $execset_1 \cdots execset_n$**)**. By scheduling in parallel I mean that the code associated with the execution sets involved can be interleaved in any way.

Loops will be represented by S-expressions whose heads are either **sloop** or **ploop**, designating serial and parallel loops, respectively[1]. The tails of the loop expressions are just execsets, implicitly a **series**.

In this notation, I can represent the execution set corresponding to execution orders (3.2) and (3.3) by something like

```
(series BB 1
        (sloop
          (parallel
            (series BB 3 BB 4)   ;update min
            (series BB 5 BB 6))  ;update max
          BB 7))
```

Notice that the explicit representation of the parallelism of the updates of **min** and **max** didn't add an exponential amount of space to the representation of just those updates, thanks to the use of the **parallel** expression. Indeed, one can encode exponentially large numbers of permutations just by giving a list of the elements (i.e., a linear amount of space on the original mass) and using those execset operators.

Finally, the serial loop above can be distributed over the two independent parts of its body, which can be represented by:

```
(parallel
  (sloop BB 3 BB 4)
  (sloop BB 5 BB 6)))
```

Now, we have the freedom to execute the loop as either a serial loop with parallel parts, or as the parallel of two serial loops. This sort of freedom is represented by the notation **(choice** $execset_1$ $execset_2$ $\cdots$ $execset_n$ **)**. The example above yields:

```
(series
    BB 1
    (choice
       (sloop
         (parallel
```

[1] In a serial loop, no iteration begins before the previous one is finished. In a parallel loop, no iteration synchronizes at all with the rest, but there is a barrier at the end of the loop. There are other possible interpretations for parallel loops (e.g., iterations may be allowed to overlap in specified ways), but they can be expressed using just **series**, **parallel**, **sloop**, and **ploop**.

```
            (series BB3 BB4)   ;update min
            (series BB5 BB6)) ;update max
        (parallel
          (sloop BB3 BB4)
          (sloop BB5 BB6)))
    BB7))
```

To say that the execution set of a piece of code is a `choice` of other execution sets means that to execute the piece of code successfully, it is necessary to execute exactly one of the `choice`s.

3.4. A Calculus of Execution Set Composition

An important feature of the execset notation is that it is possible to calculate with it. In particular, considerations of inclusion or relative size of execution sets can be made without reference to the elements of the sets themselves.

For instance, assume that you are given the execset expression `(series A (series B ...))`. This expression represents those permutations of {**A**, **B**,...} that are series in which first **A** is executed, and then a series is executed, consisting of **B** and the rest. Well, we can reason that, independently of **A**, **B**, etc., the expression above is nothing but the series that involves all those masses in the same order. We can represent this conclusion by a simple rule:

```
(series A (series B ...))
            ⇔
      (series A B ...)
```

We can use this rule to simplify an execset expression that contains that nested usage of `series`. Although this rule is established by consideration of the meanings of the execution set expressions involved, its application is completely syntactic in nature: once the rule is established, it can be used as a lemma in future simplifications, without having to resort again to analyzing the meaning of the particular expressions involved.

3.4.1. Semantics of Execution Set Notation

Here I give a more formal presentation of the semantics behind the Execution Set notation. The intention is to justify the mechanics of the calculus.

3.4.1.1. Paths

Let S be a finite set $\{s_0, s_1, ..., s_n\}$. A *path* through S is a permutation of a subset of S. More precisely, if

$$(\exists k \leq n)\ p = (s_{i_0,}\ s_{i_1,}\ ...,\ s_{i_k}) \quad \text{where each } s_{i_j} \in S, \text{ with no repetitions.} \tag{3.4}$$

then $p \in Paths(S)$.

I now define some operations that construct paths from paths. Let S_1 and S_2 be finite sets, with typical elements s_{1i} and s_{2i}, respectively. Let also

$p_1 = (s_{1i}),\ 1 \le i \le k_1,\ k_1 \le card(S_1)$

and $p_2 = (s_{2j}),\ 1 \le j \le k_2,\ k_2 \le card(S_2)$ be paths through S_1 and S_2.

The *disjoint sum* of p_1 and p_2 is the concatenation of both, in the same order, and is defined by:

$$p_1 \oplus p_2 = (s_{11}, \cdots, s_{1k_1}, s_{21}, \cdots, s_{2k_2}) \tag{3.5}$$

The disjoint sum of two paths happens to be a path too: it is clear that $p_1 \oplus p_2$ belongs in the paths of the disjoint union of S_1 and S_2 (whence the qualification of *disjoint* sum). Other properties: The empty sequence is an identity for this operation. Disjoint sums are associative (but not commutative).

The definition extends to sets of paths: the disjoint sum of two sets of paths is the set of the disjoint sums of pairs of paths, taken one from each of the given sets.

A *disjoint product* of p_1 and p_2 is any proper interleaving of their parts, as defined by:

$$\begin{aligned} &p_1 \otimes p_2 = (t_1, t_2, \ldots, t_{k_1+k_2}) \\ &\text{where } t_k = s_{ij} \Leftrightarrow (\forall g,\ 1 \le g \le j)\ (\exists h,\ 1 \le h \le k)\ |\ t_h = s_{ig} \end{aligned} \tag{3.6}$$

Although there is only one disjoint sum of two paths, there are many disjoint products (actually, $\begin{bmatrix} k_1+k_2 \\ k_1 \end{bmatrix}$ in total).

The empty sequence is again an identity for this operation.

The definition again extends to sets of paths. The disjoint product of two sets of paths is the set of the disjoint products of pairs of paths, taken one from each of the given sets.

Most uses of disjoint products in what follows refer to the properties of the set extension. So, a bit of notation is required: when in a disjoint product I use the name of a path, it should be understood that the singleton set of that path is actually used. So, in what follows, I take disjoint sums and products to be defined on sets of paths and yielding sets of paths.

The proof of the following theorem demonstrates these conventions.

Theorem 3.1

Disjoint products are associative.

Proof.

Consider $p \otimes (q \otimes r)$. A representative of this set of paths will be a path that contains elements from p, q, and r. Define a mapping from sequences to sequences that, given the representative chosen above, will return a sequence, call it pq, that ignores the elements of r. Notice that (1) this sequence is a path, indeed an element of $p \otimes q$, and (2) the representative sequence is a result of interleaving the elements of r into pq. Thus, the representative belongs in $(p \otimes q) \otimes r$ too.

The other inclusion mirrors this argument.

□

Disjoint products and sums do not distribute. Instead, we have

Theorem 3.2

$((p\otimes q)\oplus r)\cup(q\oplus(p\otimes r))$ is a proper subset of $p\otimes(q\oplus r)$

Proof.

We prove this for the case of singleton arguments, as that is enough.

Every element of the right hand side can be seen as the interleaving of elements taken from p into a concatenation of q and r. There are three classes of distributions of the elements of p into the concatenation. First of all, the elements of p could be interleaved among the elements of q, leaving the elements of r uninterrupted at the tail of the interleaving. These distributions are all members of $(p\otimes q)\oplus r$. A second class leaves the elements of q contiguous, but interleaves all the elements of p among the elements of r. These distributions are all members of $q\oplus(p\otimes r)$.

The elements in either of the above classes (i.e., in their union) are all interleavings of p into the concatenation of q and r. This shows that the left hand side is a subset of the right hand side.

On the other hand, there are interleavings of p into $q\oplus r$ that leave some elements of p among the elements of q, and the rest among the elements of r. Those interleavings form the third class, which is non-empty when p, q and r are paths with more than one element. Thus the inclusion is proper.

□

Notice that the disjoint sum of two paths is just another of their interleavings. So, when dealing with sets of paths A and B, $A\oplus B$ is a proper subset of $A\otimes B$.

If all the sets involved are the same, the disjoint sum and product have a especially simple form that deserves its own notation. The *scalar multiple* of a path p by the natural number k is defined by:

$$\begin{aligned} 0\cdot p &= () \\ k\cdot p &= p\oplus((k-1)\cdot p) \end{aligned} \tag{3.7}$$

Similarly, a kth power of a path is defined by:

$$\begin{aligned} p^0 &= () \\ p^k &= p\otimes p^{k-1} \end{aligned} \tag{3.8}$$

These definitions extend to sets of paths too, in the natural way. In that extension we see also that the kth multiple of a set is a subset of the kth power of the same set.

3.4.1.2. Execution Set Meanings

We can use the disjoint sum and product operations to define formally what each of the execution set forms mean.

execset
: The form **(execset** *f***)** stands for the subset of *Paths*(*Mass*(*f*)) that is compatible with the flow dependences. This form is called a *trivial* execset, as it does not encode any information beyond the definition of execution set. It is therefore a base case for structural induction when dealing with execsets.

series
: The form **(series** e_1 e_2 $\cdots$ **)** stands for the disjoint sum of the sets defined by e_1, e_2, $\cdots$.

parallel
: The form **(parallel** e_1 e_2 $\cdots$ **)** stands for the disjoint product of the sets defined by e_1, e_2, $\cdots$.

sloop

$$\textbf{(sloop e)} = \bigcup_{k=0}^{k=\infty} k \cdot e \tag{3.9}$$

ploop

$$\textbf{(ploop e)} = \bigcup_{k=0}^{k=\infty} e^k \tag{3.10}$$

choice
: A choice between execsets stands for the (disjoint) union of the sets represented by its subforms.

3.4.1.3. Simplification by identities

These definitions can be used to prove the validity of a collection of simplification identities, as well as to guide the discovery of other relationships. For instance, the simplification of **(series** *A* **(series** *B C***))** to **(series** *A B C***)** is a consequence of the associativity of the disjoint sum. Also, the fact that the execution set denoted by **(parallel** *A* **(series** *B C***))** is larger than the one of **(choice (series (parallel** *A B***)** *C***) (series** *B* **(parallel** *A C***)))** is a consequence of the limited distributivity of the disjoint product into the disjoint sum.

In presenting these identities, I will write **x . y** whenever I mean the list whose first element is **x** and whose rest is the list **y**, (i.e., **(cons x y)**), and **x | y** whenever I mean the list whose elements come from the lists **x** and **y**, listed in that order, i.e., **(append x y)**. Also, variables like **?x** stand for anything that could match part of a pattern.

Trivial Nests

N1: **(series ?A) ⇔ ?A**
N2: **(parallel ?A) ⇔ ?A**
N3: **(choice ?A) ⇔ ?A**

These are especially obvious identities. Essentially, if previous simplifications have reduced a expression to a condition in which there is only one thing to do, that thing gets done without further qualification. Of course, this doesn't apply to loops, as in a loop there may be only thing to do, but it will be done possibly more than once.

Loops

L1: `(sloop (series . ?A))` ⇔ `(sloop . ?A)`
L2: `(ploop (series . ?A))` ⇔ `(ploop . ?A)`

These depend on noticing that the bodies of loops are **`series`** by definition.

Non-trivial Nests

O1: `(series ?X)|((series . ?Y) . ?Z))` ⇔ `(series ?X|?Y|?Z)`
O2: `(parallel ?X)|((parallel . ?Y) . ?Z))` ⇔ `(parallel ?X|?Y|?Z)`
O3: `(choice ?X)|((choice . ?Y) . ?Z))` ⇔ `(choice ?X|?Y|?Z)`

These are generalizations of the example at the beginning of this section. Essentially, if a **`series`**, **`parallel`**, or **`choice`** expression is immediately embedded into another, the inner one can be spliced into the outer one without damage. These generalizations follow, respectively, out of the associativity of the disjoint sum, the associativity of the disjoint product, and the associativity of the disjunction in propositional logic.

3.4.1.4. Simplification by inequalities

The results of Section 3.4.1.3 apply to the simplification of an execution set expression to another that stands for exactly the same execution set. There are also simplifications that are possible only when an execution set is known to be a proper subset of another.

First we define a partial order on execution set *expressions*, namely the order induced by inclusion among execution sets. Let es_1 and es_2 be two execution set expressions, and let E_1 and E_2 be the execution sets they stand for. Then es_2 *subsumes* es_1 if and only if E_1 is a proper subset of E_2. This I write: $es_1 < es_2$.

As examples, notice that the observation that *k*th multiples are subsets of *k*th powers means directly that the execution set represented by `(sloop (execset X))` has to be smaller than the execution set represented by `(ploop (execset X))`. Thus, for all X, we can write:

`(sloop (execset X)) < (ploop (execset X))`

and, with more reason

`(series . ?X) < (parallel . ?X)`

These *inequalities* in execution set expressions find application in the elimination of redundant branches in **`choice`**s. A branch in a **`choice`** is *redundant* if and only if it is subsumed by another branch. Removing a redundant branch does not change the execution set represented by the **`choice`** expression.

For instance, assume that an execution set is represented by a choice between `(sloop (ploop (execset 1))(sloop (execset 2)))` and `(sloop (ploop` `(execset 1)) (ploop (execset 2)))`. As the execution set `(sloop (execset 2))` is just a subset of `(ploop (execset 2))`, there is no damage in discarding completely the first interpretation.

So, we have two more identities:

```
P1: (choice (series . ?X) (parallel . ?X) . ?Y)
      ⇔ (choice (parallel . ?X) . ?Y)
P2: (choice (sloop . ?X) (ploop . ?X) . ?Y)
      ⇔ (choice (ploop . ?X) . ?Y)
```

In **P1** and **P2**, the rules hold also if the order of the **series** and **parallel** forms is permuted, of course. More generally, the order of execution sets that are parts of a **choice** expression is irrelevant, and so we can reorder those expressions if so desired.

3.4.1.5. Reduced Forms

The identities and inequalities of Sections 3.4.1.3 and 3.4.1.4 show that a given execution set can be represented by many equivalent expressions, the equivalence arising essentially from syntactic redundancy[2]. In order to permit efficient comparison of execution sets, their expressions should be stored in a common form that does not suffer from this redundancy. (The Transformer of Chapter 5 will have abundant need to compare execution sets.)

To this effect, I will use the identities and inequalities described above to define transformations of execution set expressions. The transformations will be named after the identities and inequalities themselves: **N1**, **N2**, **N3**, **L1**, **L2**, **O1**, **O2**, **O3**, **P1**, and **P2**. In each case, the transformation is defined this way: an execution set expression that matches the left hand side of a transformation can be replaced with the right hand side, properly instantiated to reflect the match.

I say that an execution set expression is in its *reduced form* when recursive application of the transformations **N1**, **N2**, **N3**, **L1**, **L2**, **O1**, **O2**, **O3**, **P1**, and **P2** cannot modify it any further. Let's refer to this process as *Algorithm R*.

Also as the order of branches in a **choice** is irrelevant, I assume that transformations **P1** and **P2** are free to reorder their **choice** patterns in order to establish the match.

[2] This redundancy is almost trivial, as it can be judged by inspection of the execution set expressions alone. On the other hand, the redundancies discussed in Section 5.5.1 can be detected only by considering the arrangement of parts of the specific program at hand, and therefore demand a comparison of the *meanings* of the execution set expressions, in the context of a given program. Thus the distinction between *syntactic simplification* (which involves only the *forms* of the expressions) and *semantic simplification* (which demands the *meanings* of the expressions).

Theorem 3.3

Each execution set expression has a unique reduced form, except for the order of choice branches.

Proof.

First of all, let's observe that *Algorithm R* is indeed an algorithm. Given an execution set expression *e*, each of the transformations **N1**, **N2**, **N3**, **L1**, **L2**, **O1**, **O2**, **O3**, **P1**, and **P2** reduces the number of parentheses in *e* by exactly 2. So, either *e* is converted into an expression with fewer parentheses, or it is a fixed point of the transformation set. In the former case, *Algorithm R* cannot iterate indefinitely, as the number of parenthesis pairs would reduce at each iteration.

Now, observe (by case analysis) that when any two of these rules apply, the application of either does not preclude the (eventual) application of the other. So, we are free to apply them in any sequence, and in as much as we keep doing it while possible, the resulting execution set expressions will be the same. (Except, perhaps, for the irrelevant order of branches of a **choice**). This is, the set of transformations supplied to *Algorithm R* has the Church-Rosser property.

Thus, the procedure defined by *Algorithm R* terminates for any execution set expression *e* by yielding a possibly different execution set expression *r*. and *r* is unique up to the order of **choice** branches.

□

3.5. Discussion

The motivation behind the development of the execution set calculus is simple: to represent abundant parallelism compactly, and in a way that can be manipulated symbolically at low cost.

Having related execution set size with parallelizability, I will show later how one can structure the parallelism detector around the theme of enlarging execution sets. The application of the theory presented in this chapter will be seen in the Transformer of Chapter 5.

As I said above, the motivation for this development was the efficient symbolic manipulation of compact parallelism representations. These desires can conflict. For instance, simple regular expressions can be used to represent flow structures (for instance, as in Path Expressions [CaH74,Cam76], or in the work of Wegman [Weg83]). Adding to this notation the closure of shuffles [Jed87] increases the power of the notation, but also increases the recognition (and, therefore, the manipulation) complexity of the generated language to context sensitive [Jed88]. Further adding explicit locks to represent arbitrary synchronization provides a very detailed representation of parallelism, but at the cost of raising the complexity of the language generated by these expressions all the way to recursively enumerable [Sha78].

Execution set expressions strike a balance, in remaining easy to manipulate while providing detail that parallelizers need. Nested **ploops** cause them to generate a hierarchy of context sensitive languages (the argument follows those of [Jed87, Jed88] with respect to general shuffle operations), but the resulting power doesn't prevent us from calculating with the resulting expressions.

In reaching this balance, some detail is lost. For instance, execution set expressions don't represent conditionals, nor do they represent explicit synchronization, nor can they be used to represent certain architectural constraints (like size of the vector register file in vector machines).

Some of this detail has been dispensed with because it can be recovered at will from other information available to the compiler. For instance, conditionals could map nicely to alternation operators (as in Flow Expressions [Sha78], for instance), but I had a more useful employment for alternation, in the **choice** operator. That doesn't prevent the parallelizer, as will be seen later, from using knowledge about conditionals in parallelization: conditionals can be detected and recovered easily from the flow graph, which is part of the assumed input to the parallelizer.

The lack of explicit synchronization comes from the desire to avoid committing to a specific synchronization mechanism (say, semaphores), as this would bring in too much architectural detail to the analysis. However, there is no way to avoid representing *some* synchronization, and it is done implicitly at the end of **parallel** and **ploop** operations.

Similarly, the lack of mechanisms to represent some architectural constraints on parallelism is necessary to avoid contaminating the detector with exploitation issues. For instance, vectorizations have the form **(ploop** *<expression>***)**, which does not show how many vector registers are effectively usable. This is a runtime feature, best left to the back end.

Finally, in spite of my best wishes to avoid committing to detail too early, some model of parallel execution is required behind the syntax. The execution set expressions embrace the well-known assimilation of parallelism to arbitrary interleaving of sequential actions.

3.6. Related Work

Describing parallelism explicitly, and trying to quantify what it means for a piece of code to be 'more parallel' than another, is central to any systematic parallelization efforts. So, every parallelization project has had to define a measure of increasing parallelism.

While, in most cases, the metric is only implied by the rest of the work (for instance, the number of dependences carried by a loop can be used to show that a transformation exposed some parallelism), some attempts have tried to describe formally what it means for an algorithm to be 'more parallel'.

For instance, Amir and Smith [AmS86] discuss parallelism in CREW models. There, nesting of composition is related to synchronization, and therefore shallower nesting means more chances for parallelism. Their work is restricted to the primitive recursive functions.

Shaw [Sha78] describes a family of languages based on regular expression abstractions of flows (of data, of control, ...) in systems. His Flow Expressions are very powerful (every recursively enumerable set can be encoded as one), and thus they may be hard to manipulate symbolically. Execution Set expressions can be considered a very restricted subset of Flow Expressions, chosen for fit to the task at hand and ease of manipulation. In addition, Execution Set expressions map alternation (in regular expressions) to choice (don't-care non-determinism), while Flow Expressions normally interpret alternation as a representation for conditionals. Flow Expressions distinguish finite iteration (possibly unbounded) from guaranteed endless looping. This distinction adds complexity and would not be very fruitful in the context of this work.

The disjoint product is essentially equivalent to the *shuffle* operator. A good survey can be found starting from [Jed87, Jed88] and the references therein.

Milne [Mil85] describes a mathematical framework to analyze concurrent systems. His system is non-deterministic. His contribution is a formal calculus of concurrency.

Time, and concurrency, can be given formalizations in logic. See, for instance [Gol87].

Finally, Lamport [Lam86] has studied formalizations of parallelism that do not demand serializability, nor even atomicity. Perhaps a different parallelization model could be designed around these ideas, but it would likely require a different analysis strategy. (As I said above, the advantage of reducing parallelism to interleaving is that this reduction permits the use of classical techniques for the analysis of sequential programs.)

CHAPTER 4

Bottom-Up Structure: The Fragment Enumerator

So far we have tools to characterize our freedom to schedule pieces of a program in different orders. However, we have no notion of what pieces of a program it could be interesting to analyze.

Choosing the program decomposition properly is critical. First of all, there are far too many subsets of a program for one to consider seriously exploring all, or even a large number, of them. Second, not all subsets of a program are 'interesting'. The notion of interest here is twofold: (1) selecting a subset of the program is a form of abstraction, some abstractions capture the meaning of the program more usefully than others; (2) empirically, we see that potential for parallelism is often found in some places (for example, loops), which suggests that special effort be dedicated to their analysis.

The goal here is to find a definition of *fragment* that satisfies as much as possible conditions like (1)–(3) of §1.2 and for which the computation of flow attributes can be accomplished efficiently.

There are many conceivable definitions of fragment. Compilation technology has already accumulated some experience with various classes of regions, like intervals [AlC76, HeU72, Tar74] ([Bur87] provides a nice summary of the state of the art), hammocks [Kas75, Kas73] and other similar notions [Sha80, Wad80]. I have selected *depth-first-numbering intervals* as my definition of fragment, and examples from here on will assume it. However, it is important to distinguish the general notion of *fragment* from its instantiations. The model works in terms of fragments, that nest to reflect the semantic of the source language. Practically all the details that follow can be transcribed mechanically to use some other definition of fragment.

4.1. Depth-First-Numbering Intervals

Given a directed graph, its nodes can be numbered in the sequence of their preorder visit, starting from some root node. These numberings are called *depth-first numberings*, because they correspond to the order in which the graph would be visited during a depth-first search. (The standard reference to why depth-first numberings are useful is [HeU75]) I will use the abbreviation *dfn* to refer to such numberings, so $dfn(i)$ refers to the node whose depth-first number is i and $dfn^{-1}(p)$ refers to the depth-first number of the node named p.

Consider a graph $G = (V, E)$ with n nodes, and think of the nodes whose dfn numbers k belong in the interval $0 \le i \le k < j \le n$. For some choices of i and j it will be true that that set is closed with respect to the original graph, in the sense that

$$(\forall\, k \text{ s.t. } 0 \le i \le k < j \le n)\ (\forall\, p \in V)\ \left[(dfn(k), p) \in E \Rightarrow i \le dfn^{-1}(p) \le j \right] \quad (4.1)$$

In this case, we call that set a *depth-first-numbering interval*, or *dfn-interval* for short. A dfn-interval is notated by listing its extreme depth-first numbers separated by a semicolon, thus: $i;j$. Such interval is said to be of *size* $j-i$.

The intuitively important characteristic of dfn-intervals is that all the arcs that start in a node in a dfn-interval $i;j$ end either in j, or in another node of $i;j$. Notice that $dfn(j)$ does not belong in $i;j$.

As an example, consider Figure 4.1, which shows the dfn-intervals of of the program graph for the **minmax** program. These intervals are summarized in Table 4.1.

The three intervals named i_5, i_6, and i_7 contain just one basic block each. They don't abstract much interesting structure, as they coincide directly with one unit of granularity of this analysis. On account of this observation, it could seem that we could ignore all such size-1 intervals. However, there is one case where an interval of only one basic block is interesting: the fragment composed of a block that is a loop in itself. Abstracting the loop into a single fragment is a valuable operation, so we cannot disregard the size-1 intervals completely. We arrive then at a compromise that keeps the useful size-1 fragments and discards those that don't abstract any interesting control structure:

Depth-first Interval Fragments. Let $i;j$ be a *dfn*-interval of graph G, as above. Then, $i;j$ constitutes a Depth-First interval fragment of G (abbreviated *dfi-fragment*) if and only if:

$$(\text{size}(i;j) > 1) \vee (i \in \text{succ}(i)) \quad (4.2)$$

The first term in 4.2, $(\text{size}(i;j) > 1)$, represents the non-triviality of abstracting as a unit some part of the program that contains more than one fragment. The second term, $(i \in \text{succ}(i))$, captures the non-triviality of those intervals that contain a single fragment that happens to be a loop in itself.

4.2. Aspects

The insight behind fragments is that certain subsets of the entire program can be considered as indivisible units for purposes of analyzing larger ones. This is a form of control abstraction. The motivation for choosing one definition of fragment over others is that certain control abstractions are especially useful towards the goal of finding parallelism.

For instance, dfi-fragments abstract some control structures at the level of basic blocks. We can even give simple descriptions of each fragment in the **minmax** example (see Table 4.2). We say that f_2 is the body of the loop, because f_2 abstracts the structure that is iterated. In Table 4.2, the intervals from Table 4.1 that are also dfi-fragments are renamed for i_k to f_k, to emphasize that only some of the intervals are also fragments under this definition. That such characterizations of function are easy to find supports informally the impression that this particular set of abstractions can

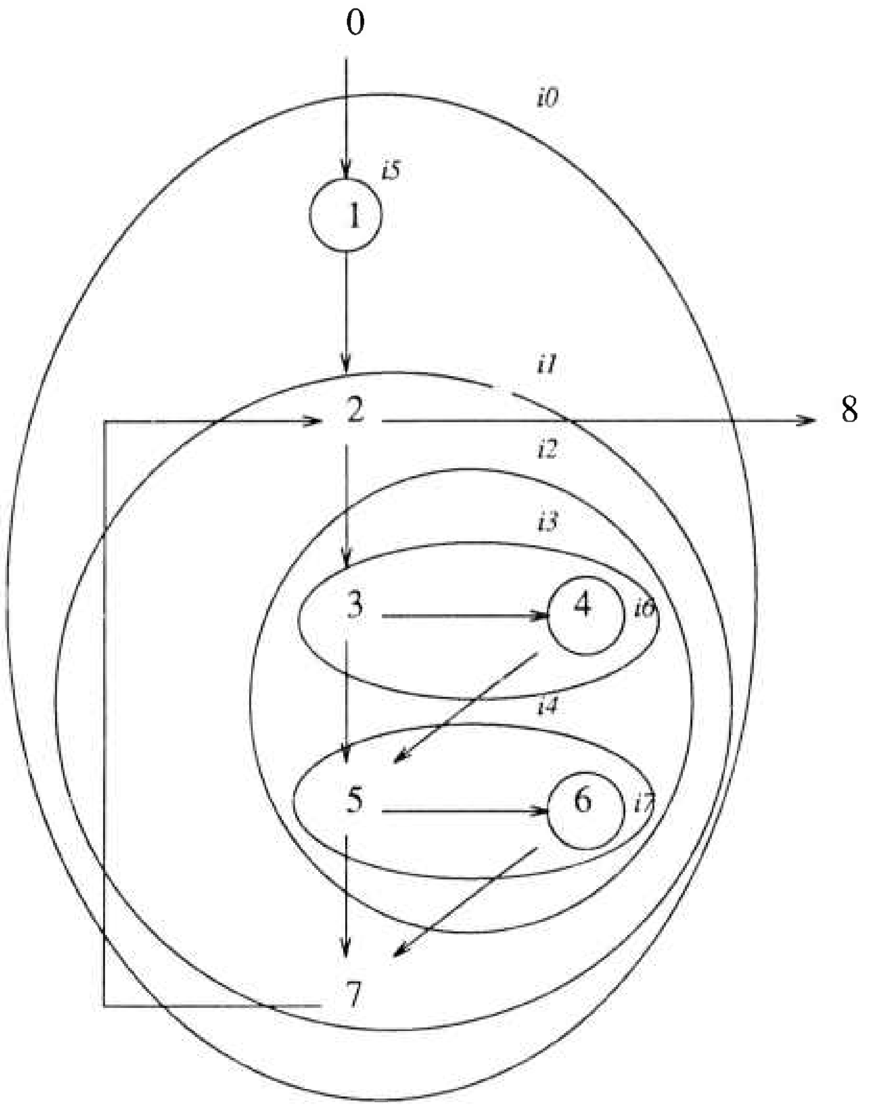


Figure 4.1. *dfn*-intervals for Minmax

Size	Name	Entry; Exit
1	i_7	6;7
	i_6	4;5
	i_5	1;2
2	i_4	5;7
	i_3	3;5
4	i_2	3;7
6	i_1	2;8
7	i_0	1;8

Table 4.1. Intervals of the Minmax Program

Size	Name	Entry; Exit	Description	Components
2	f_4	5;7	Update of Maximum	
	f_3	3;5	Update of Minimum	
4	f_2	3;7	Body of the Loop	f_3, f_4
6	f_1	2;8	The Loop	f_2
7	f_0	1;8	The Program	f_1

Table 4.2. Fragments of the Minmax Program

be useful to understand the program starting from its parts. (The ultimate test of this vague impression is, of course, that the we can write useful parallelizations in terms of this analysis. Chapter 6 presents my experiences in this respect, but some informal motivation here may be useful.)

As I said before, considering all combinations of subsets of a program is too expensive. Even considering all combinations of fragments can be too expensive. I need to formalize the idea that only some analyses of a fragment in its parts can be used for parallelization, if we want to keep the running time manageable. My notion of *aspects* of a fragment captures the idea of restricting the parallelization effort to the interactions of only some combinations of fragments.

This section presents the terminology needed for this discussion. I will show later, in Chapter 6, that the restrictions imposed by my definitions of fragment and aspect, while pruning drastically the total amount of work with respect to a naive all-subsets analysis, still manage to enable all the useful transformations from the literature.

Table 4.2 introduces the notion of *compositions* of a fragment. Fragment f_4, for instance, is a maximal subset of fragment f_2, thus it is listed as a component of the latter. Although f_4 is also a subset of f_1, it is not maximal, and so it is not listed. Some definitions are needed here.

Component. Fragment f is a component of fragment g if and only if f is a proper subset of g.

Maximal Component. Fragment f is a maximal component of fragment g if and only if f is a component of g, and f is not a component of another component of g.

Fragment Composition. A composition of a fragment f is a collection $f_1, f_2, \ldots$ f_k of maximal components of f. Compositions are not necessarily exhaustive covers.

Although fragments abstract control structure, they are not totally devoid of control structure themselves. Composition formalizes the notion that further abstraction is possible, this time at the fragment (instead of basic block) level. Such abstraction collapses entire fragments down to single nodes.

Aspect. Let f be a fragment of G and let $C = (f_1\ f_2\ \cdots\ f_k)$ be a composition of f. The aspect of f under collapse of C, represented by $[f/C]$, is the flowgraph that derives from that of f by substituting a new single node $\overline{f_i}$ for all the nodes in each f_i. This substitution converts arcs entering f_i into arcs entering $\overline{f_i}$, and arcs leaving f_i and entering $f-f_i$ into arcs leaving $\overline{f_i}$.

By collapsing each component of a fragment to a single node, we obtain alternative views of the flow inside that fragment. Consider for instance Figure 4.2. There I depict four compositions of the fragment f_2 (which is dfn-interval 3;7).

I will represent aspects by giving the fragment being considered and the list of the fragments being collapsed. So, 3;7 considered without any internal collapse will be denoted by [3;7/()], while 3;7 with fragments 3;5 and 5;7 collapsed to single nodes will be denoted by [3;7/(3;5 5;7)].

Aspects can then be used to expose high-level control structures. In the view of 3;7 provided by [3;7/(3;5 5;7)] we can discern a very important structure: regardless of the internals of 3;5 and 5;7, fragment 3;7 is just a simple sequence of those two maximal components. I will use this to obtain a concise encoding of parallelization

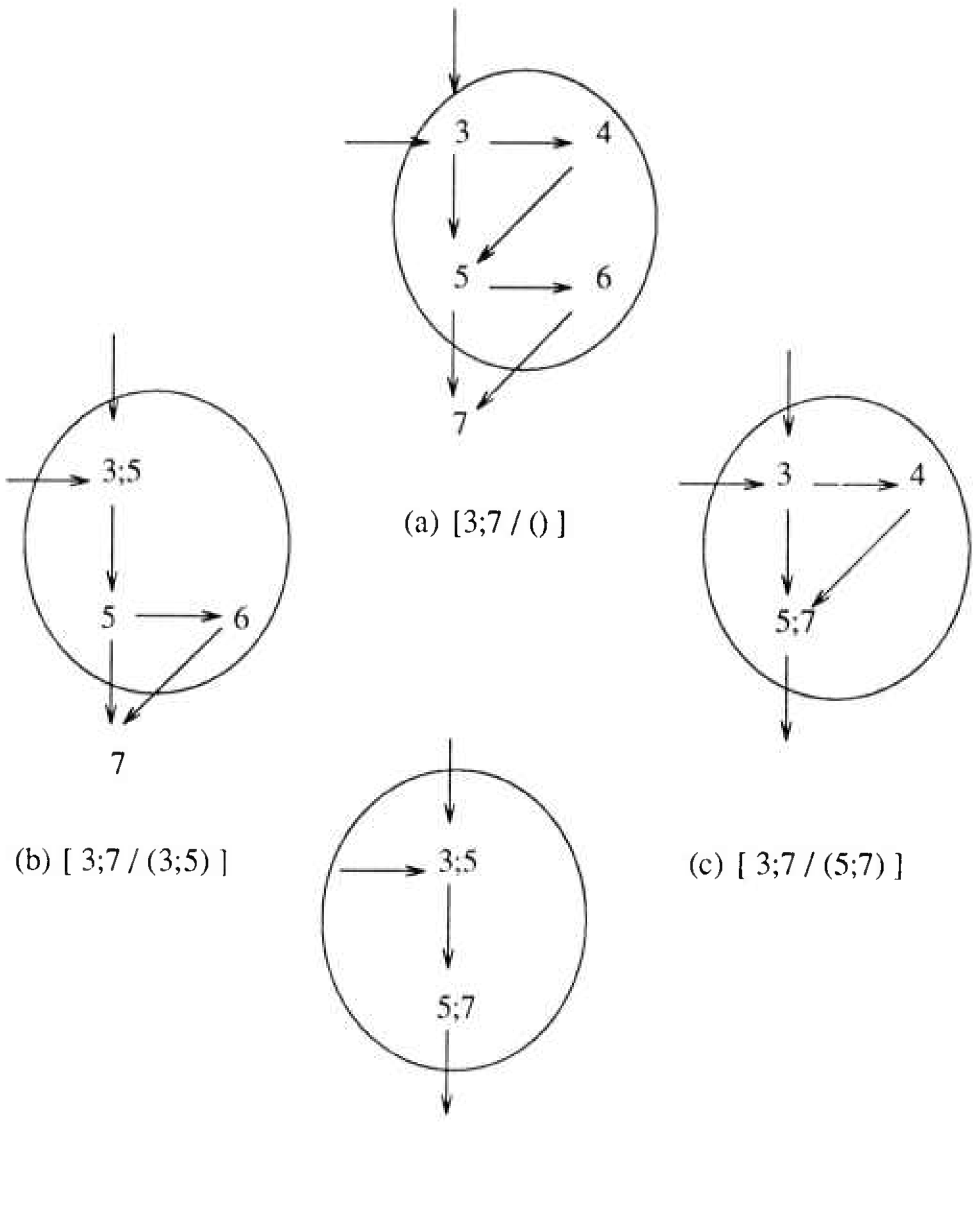


Figure 4.2. Aspects of the Loop Body

conditions: I need to provide only a single rule for parallelizing sequences, and will depend on aspect formation to expose to the same transformations sequences of higher-level (i.e., sequences composed of fragments more complex than single granules).

4.3. Aspect Formation

The set of aspects of a given fragment hierarchy defines a relationship between each fragment and some sets of its subfragments. I use this relationship to propagate the results of the analysis upwards from simple fragments to their super-fragments. The character of this relationship is that an aspect essentially records an attempt to parallelize a fragment by eliminating (hopefully) irrelevant detail in *some* of its subfragments.

The idea is to take the (static) hierarchy of fragments, and visit them from small to large, enumerating for every fragment all the aspects that involve only fragments that have been visited already. This involves first selecting for each fragment which subfragments will be used to construct aspects, and then ordering the visits after a topological sort of the hierarchy, where the partial ordering is fragment inclusion.

For each fragment f, let $Components_1(f)$ and $Components_2(f)$ stand for subsets of the subfragments of f.

For each subfragment s in $Components_1(f)$, we construct an aspect for f by calculating the quotient $[f/s]$. After each such quotient, we consider the nodes that remain uncollapsed in the new graph (i.e., the nodes in $f-s$). We then list the members of $Components_2(f)$ that *fit* in the remaining mass. These are the subfragments all whose nodes and edges are members of the nodes and edges of the remaining mass. Let s_1 be one of them. Then we collapse s_1 in $f-s$, and repeat the procedure, starting now with $f-s-s_1$. When no more fragments fit in the remaining mass, we have a components list and an aspect.

I have left unspecified what subfragments of f are in the sets $Components_1(f)$ and $Components_2(f)$. This is another design decision, exactly like the definitions of granule and fragment. The choices range from considering all the subfragments of f as *Components* to choosing only a small number of them. The first of these sets determines which subparts of a fragment start the aspect formation process; the second set determines which subfragments are deemed available for further collapse.

SPOIL uses the following strategy for aspect formation:

$$Components_1(f) = Components_2(f) = Maximals(f) \qquad (4.3)$$

where $Maximals(f)$ is the set of maximal subfragments of f.

4.4. Iterators

We now have a fragment hierarchy, and a set of aspects (determined by our choices of $Components_1$ and $Components_2$). Now we want to organize the analysis in such a way that the eventual enumeration order ("small ahead of large") be obtained. To this end, rank all the fragments according to the following procedure:

(1) First, in any order, all the fragments f for which $Subfragments(f)=\varnothing$. There have to be some fragments that satisfy this condition, because there have to

be minimal fragments. If f is a subfragment of a fragment g, remove f from the components of g.

(2) Repeat (1) until no unranked fragments are left.

The ranking thus obtained represents a topological sort of the fragments, that puts first the fragments that have no structure, and then successively larger fragments, in order of their inclusion relationship.

If we now enumerate the aspects of every fragment, in the order of their ranking, we will have the guarantee that every fragment is enumerated eventually after all its subfragments. This is the *eventual enumeration ordering* property.

I capture this notion in SPOIL by means of *iterators*. An iterator is a function of a fragment hierarchy, a fragment, and a list of fragments, that returns a list of aspects. Its meaning is this: $iterator(fh,f,history) = visits$ if and only if $visits$ is the list of aspects that are needed to enumerate f completely, assuming that the nodes in *history* have been visited before.

The search for parallelizations occurs in this structured space of aspects: the Fragment Enumerator of Figure 1.1 is essentially an iterator for the fragment that comprises the entire program. For each aspect returned by the iterator, the Transformer is asked to parallelize it, probably using the results of having already seen all its subfragments, according to the eventual enumeration ordering.

4.5. Consequences of the Bottom-Up Organization

Please consider again Figure 4.2. Fragments f_3=3;5 and f_4=5;7 correspond to the updating of the running values of the minimum and maximum found so far. The figure shows alternative views of the fragment f_2=3;7. For instance, part (d) shows that f_2 can be read as a simple sequence of nodes. Fragment f_3 abstracts basic blocks *BB* 3 and *BB* 4; while f_4 abstracts basic blocks *BB* 5 and *BB* 6. Similarly, the communication patterns are also abstracted: Arcs (*BB* 3, *BB* 5) and (*BB* 4, *BB* 5) merge into the new arc (f_3, f_4), representing the fact that *at this resolution*, leaving f_3 implies entering f_4.

Notice also that the two updates are "independent", in an intuitive sense: both store new values in different variables, and the values stored by one are never needed by the other, nor overwrite values the other may eventually use.

This suggests a possible rule of parallelization, which we can state informally by saying that *given two masses of code linked in sequence, they can be executed in parallel provided that (1) they don't write on the same locations, and (2) they don't read locations potentially written by the other.* (This classical rule first appeared in [Ber66].)

To exploit this idea, we could look for such a conspicuous pattern as two (or more) nodes chained together in the shape of a string. This plan raises two questions. The first is about the meaning of pattern recognition in the class of program graphs, and I address it later (Section 5.2). The second question is about the global organization of the search. That is, assuming that we have a mechanism to decide if a given graph matches a specified pattern, how do we set up the search in the space of all the possible subgraphs and all the possible patterns?

I have already advanced my solution to the second question. The hierarchy of fragments drives the search, by presenting one fragment at a time for consideration by the transformer. The guarantee of eventual enumeration ordering provides the necessary structure to the search: any interesting parallelization discovered for a fragment is eventually visible to—and can affect the parallelization of—all its super-fragments.

Consider, for instance, the last time f_2 is presented for transformation. It will be true by then that f_3 and f_4 have been presented before, so the flow information visible at their boundaries will be known, thus enabling the evaluation of the conditions for the independent sequences parallelization, as outlined above.

Similarly, when f_0 is presented as [1;8/(1;2 2;8)] we will notice that it too has a string-like structure, but now we have that 2;8 updates a variable defined by 1;2—namely, the loop index. So, the intended parallelization won't apply in this case.

4.6. Fragment Hierarchy Interface

In order to abstract the internals of the fragment enumerator, I need to assume some interface for the fragment hierarchy. I will say that each fragment has an associated key, its *fragment id*, that can be used to store it in or retrieve it from the fragment hierarchy. This is represented by the functions **`get_id(fragment)`** and **`get_fragment(id)`**.

I will also need an iteration facility on fragments. The function **`make_fragment_iterator(id)`** returns an object that keeps state about an iteration on all the subfragments of the one designated by the id. Then, the function **`get_next_fragment(iterator)`** will return either another subfragment of the one on which **`iterator`** was primed, or a distinguished value representing the exhaustion of the iterator. These iteration functions implement the eventual enumeration ordering guarantee.

4.7. Related Work

Fragment hierarchies are about abstracting detailed computations. As such, any procedural abstraction mechanism could be used to design a fragment decomposition.

More concretely related, one can find efforts to summarize the control structure of a program. For instance, Lepage et al. [LBR81] consider a multi-level directed graph, in which a single node at a given level may stand for a single-entry subgraph of the node immediately underneath. Sharir [Sha80] builds a depth-first search tree starting from the control flow graph, and then tries to reduce specific control structures.

Fragment hierarchies are also a matter of focus: the parallelizer must consider 'interesting' pieces of its input. In this sense, work that aims at identifying 'interest' in subgraphs of a program representation is related to the work reported here. For instance, consider the *Π–blocks* of [Ban79]. A *Π–block* in the dependence graph for a program is either a maximal cycle or an isolated point. Banerjee tries to construct distributed versions of the code for each Π–*block*, and so Π–*blocks* play a role similar to that of my fragments, in that they are a focus of analysis. On the other hand, Π–*blocks* don't nest (as they are maximal cycles), and so they can't be used to organize a hierarchical search. Also, the units of parallelization are then inside Π–*blocks*, thus being closer to my notion of granule than to my notion of fragment.

Warren [War84] considers a dependence-based program representation that joins the concepts of loop carried dependence and hierarchical abstraction. The idea is to keep in a single place all the information needed to reorder a loop.

Lam's [Lam88] VLIW compiler for the Warp [AAG87, KuM84] schedules loops by means of a hierarchical reduction step. The intention is to abstract away the structure imposed by conditionals, so that the inner loops appear as single blocks to the scheduler (which otherwise would inhibit its search at the conditionals). This hierarchical reduction also drives the search: the scheduling terminates when the entire program has been reduced to a single node.

All these investigations have abstracted the control structure of a program in order to progress from small to large pieces. An element that distinguishes the research reported here is the notion of aspect, which formalizes the idea of considering the same piece of a program under different abstractions of its subparts.

CHAPTER 5

Exposing Parallelism: The Fragment Transformer

The actual work of parallelism recognition occurs in the *transformer* of Figure 1.2. This chapter describes how the fragment enumeration is eventually converted into an execution set for the program as a whole.

5.1. ExecSet Hierarchy

Execution sets are defined by a mass and a partial order. A fragment represents one such arrangement, so each fragment has exactly one execution set. Thus we define a function *execset*: *Fragments* → *ExecSets* to capture this association.

However, determining *the* execution set for a fragment is equivalent to determining precisely the dependences (ergo, equivalent to solving the general parallelization problem). So we can't (in general) obtain the precise execution set for a fragment. We can, however, approximate the values of the *execset* function with conservative execution sets, where by conservative I mean execution sets that are smaller (i.e., more serial) than the ideal ones.

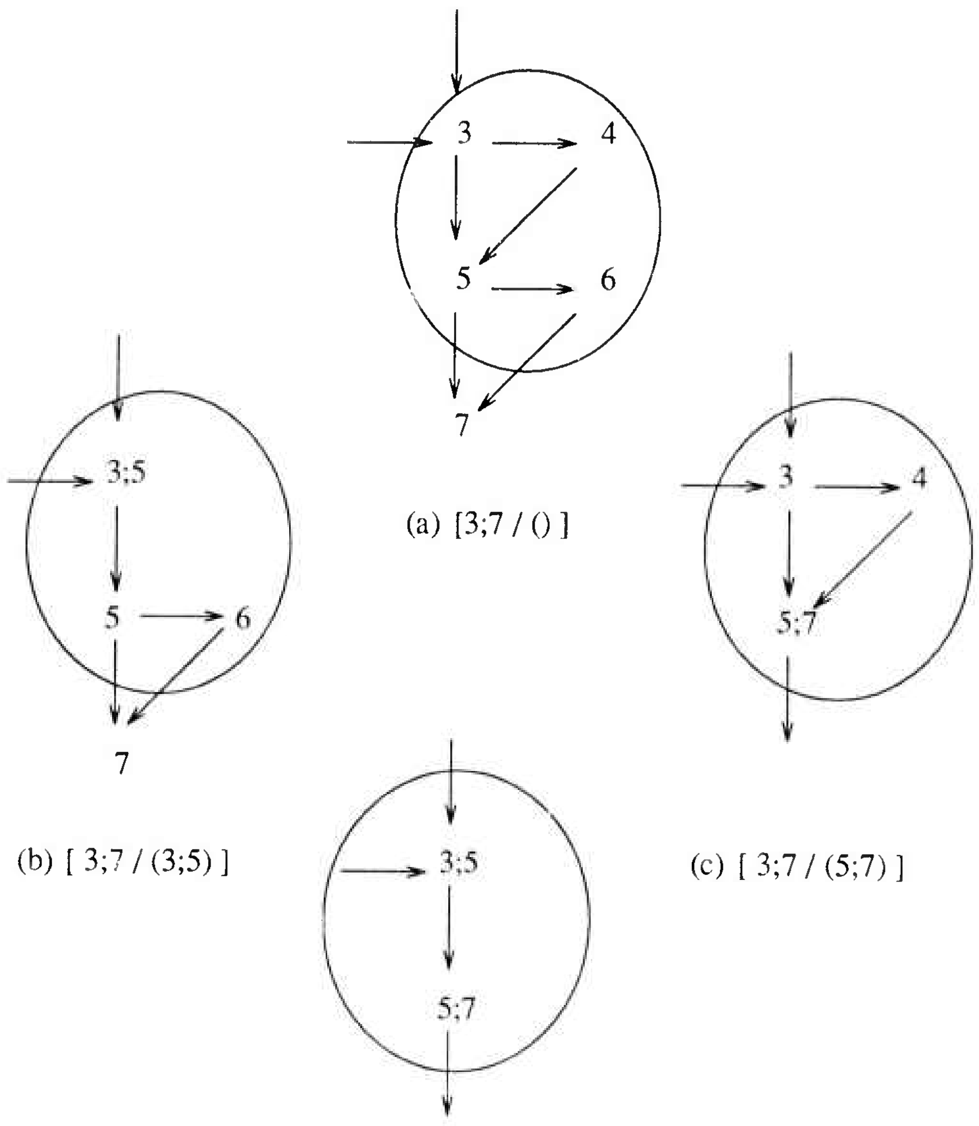
Thus we do not consider the execution set to be a static attribute of a fragment. Instead, we start with a correct (if, perhaps, overly serial) execution set for each fragment, and obtain a sequence of approximations over time, based on the results of work on sub-fragments, until we obtain the largest execution set that we can. How large this execution set is, in absolute terms and with respect to the ideal one, is relative to the power of the parallelism detection mechanism that I describe in this chapter.

The connection between fragment ids and execution sets also changes with time. Transformations that add or move code make it undesirable to tie execution sets to specific program graphs. If a transformed fragment at a given position in the program graph retains the meaning of the overall program, I want to consider its execution set as a description of a valid alternative to the original fragment.

So, as a convention, I will treat fragment *id*s as defining a *position* in the program, rather than a specific program subgraph. The idea is the following: when a fragment is transformed, the resulting execset is stored back in the execution set hierarchy under the same key (namely, the *id* of the fragment) that the original execset had. (Remember that the fragment hierarchy and the execution set hierarchy are isomorphic.) So we have two functions to access and update current execset:

```
get_current_execset(id)
set_current_execset(id, new-execset)
```

Aspect formation also causes execution sets to change over time. As an example, consider the four compositions of 3;7 shown in Figure 4.2, repeated here for



(a) [3;7 / ()]

(b) [3;7 / (3;5)]

(c) [3;7 / (5;7)]

(d) [3;7 / (3;5 5;7)]

Figure 4.2 *bis*. Aspects of the Loop Body

convenience. Although the graphs obtained by collapsing the components are different, their execution sets will all be associated with the *id* 3:7.

A consequence of this evolution over time of the execution sets is that the typical execution set expression is a disjunction of alternatives. When a parallelization provides a new alternative, it is first checked for reducibility to one of the already known ones. This is done by first computing a reduced form for each alternative. For instance, both `(series` *A* `(series` *B C*`))` and `(series (series` *A B*`)` *C*`)` reduce to just `(series` *A B C*`)`, which is the form actually stored.

5.2. Recognition of Patterns in Graphs

In Section 4.5, I left open the question of what pattern recognition in the class of program graphs is. This question has two aspects: what are we willing to call a pattern and how does one decide if a graph matches it or not.

These issues could be resolved in their generality if we had available some sufficiently powerful recognition technique. For instance, context-free graph grammars would provide a reasonable setting for a discussion of pattern matching in flow graphs.

Regrettably, the question of the maximal class of context-free graph grammars that can be parsed is still essentially open. Some subclasses have been parsed successfully, so I will argue for using some parsing mechanism, in as much as the grammars thus parsable are sufficiently expressive.

SPOIL implements a parser for Farrow et al.'s SSFG (Semi-Structured Flow Graphs) grammar. This is a single grammar, but designed specifically to capture the structure of flow graphs. The reference to this work is [FKZ76], to which the reader is referred for further details. See also Appendix C for a quick summary.

5.3. The Catalog of Parallelizations

The catalog will contain rules indicating which tests are needed to permit a given parallelization and what the results of applying it are. In the descriptions that follow, *f* stands for the fragment presented to the transformer.

5.3.1. Rule Structure

Let's begin by describing the structure of the rules. Rules will be composed of a *guard* and some *transformations*.

The guards are tests on the properties of the fragment. They are written in terms of a small number of primitive tests, organized recursively. Guards have two functions. First of all, they are predicates (*does this graph parse as a loop? do these two fragments write on the same location?*) that determine whether the rule applies. Second, they bind names to objects discovered during the testing (for instance, the parts of a parse, the data flow sets, some relevant subfragments, etc.). Those names are represented in what follows as identifiers whose first character is a question mark.

The transformations consist of two sets of effects: first, *reorganization and compensation*, which may create new entries in the fragment hierarchy and modify pieces of code to expose some parallelism; and second, the calculation of a new *execution set* to reflect the parallelization accomplished. This execution set is reduced and, if not otherwise subsumed in previous parallelizations, it is included in the alternatives list for the fragment id that is kept in the ExecSet hierarchy.

5.3.2. Guards

Guards are expressed in terms of a set of primitive tests, connected by a set of controlling connectives. The binding of variables is accomplished implicitly, when a rule takes a pattern as an argument, or explicitly, by means of the **match** primitive.

Although guards are predicates, and their expression could then be completely logical, I have studied only guards defined procedurally: guard satisfaction is described as sequential, recursive, evaluation of each term in the guard, following the left to right order in each recursion level. This definition was chosen solely for implementational convenience. Considering a guard language defined in purely logical terms remains an area open to investigation.

The primitive tests used by SPOIL are described in Appendix A; this section gives a rationale for them and provides part of the background needed for understanding the example in Section 5.4.

Grammar tests. These consist of a context-free graph grammar, against which we attempt to parse the given fragment. The result of a successful parse can be accessed by means of paths in the parse tree. For instance, define a grammar for sequences, the S-grammar, in this way:

$$\begin{aligned} \rightarrow \mathit{Sequence} \rightarrow &: \rightarrow \mathit{First} \rightarrow \mathit{Tail} \rightarrow && \text{(S-grammar)} \\ \rightarrow \mathit{First} \rightarrow &: \rightarrow \mathit{node} \rightarrow \\ \rightarrow \mathit{Tail} \rightarrow &: \rightarrow \mathit{Sequence} \rightarrow \\ &\mid \varnothing \end{aligned}$$

and let's say that fragment *f* is derived by the S-grammar in the following way:

$$\begin{aligned} \rightarrow \mathit{Sequence} \rightarrow &: \ \rightarrow \mathit{First} \rightarrow \mathit{Tail} \rightarrow \\ &: \ \rightarrow g \rightarrow \mathit{Tail} \rightarrow \\ &:^* \rightarrow g \rightarrow h \rightarrow \equiv f \end{aligned}$$

then the pattern matcher allows us to refer to g as $f.\mathit{Sequence.First}$, and to h as $f.\mathit{Sequence.Tail}$. (The syntax actually used by SPOIL is much less graceful, but equivalent.)

Fragment Structure tests. These are expressions of one of the forms:

(**exists** *guard iter fragment−id−list output*)
(**forall** *guard iter fragment−id−list*)

The meanings of these forms are as follows: An **exists** form succeeds if any fragment id in the *fragment−id−list* (which are always subfragments of *f*) satisfies the *guard*. These tests are done by successively binding the *iter* variable to the members of the list, and collecting the ones that succeed in *output*. A **forall** form succeeds if all the (current) subfragments of *f* in the *fragment-id-list* satisfy the recursive guard. Nothing is bound in a **forall**.

Data flow tests. These consist of standard data flow equations (examples appear in the rules given in Section 5.3.4). These tests succeed if the equations can be satisfied in terms of the data flow information. These tests can bind the results of some set-theoretic manipulations that are needed to satisfy the equations. For instance, if an equation requires that the intersection of two sets of used variables be non-empty, we can give a name to that set, so that the actions can refer to those variables.

Execution Set tests. These tests compare known execution sets (obtained by means of **(get-current-execset** *fragment***)**) against each other or against patterns. This matching is a simple, syntactic, unification on S-expressions. At the atomic level, we either have the distinguished constants **series**, **parallel**, **sloop**, or **ploop**, or we have other execution sets. When we descend to execution sets with no structure, matching is by identity. The patterns include unification variables (denoted here with names like **?List**), these are the names bound by a successful guard. The tests are expressed as calls to the function **match**, which performs the actual unification.

To deal with **choice** branches, I use an **exists-execset** primitive test. This test takes two arguments: a fragment id and a pattern. If the given fragment id's execset expression is a **choice** form, then each of its branches is tested against the pattern (and bindings made if any is successful). Else, the execset expression itself is matched against the pattern.

Connectives. The guard primitives discussed above are connected by the **and**, **or**, and **not** connectives. These connectives are defined in a procedural, not purely logical, way: their arguments are considered left-to-right, and the connective succeeds or fails accordingly. Success is defined as being able to produce a set of bindings for the unification variables; failure, as finding an irreconcilable contradiction. Each term is tested in an environment where all the previous terms have already left their bindings available.

And succeeds if and only if all of its terms succeed. The bindings produced by each of the terms are accumulated, and are available to the rest of the rule. However, as soon as a term fails, the entire form fails and no bindings are accumulated. **Or** succeeds as soon as a term succeeds. No other terms are tried (which means that the rest of the rule cannot depend on observing bindings produced by those terms). The bindings produced by the term that succeeded are available, the rest are abandoned. **Not** succeeds if and only if its only term fails. No new bindings can be generated by a **Not**, not even bindings arising from the partial success of its argument (i.e., if the argument succeeds in making some bindings and later fails, those bindings made before the failure are lost too).

5.3.3. Catalog Structure

In this work I assume that the catalog is *flat*. This is, it is just a bag of rules. This organization (or lack thereof) makes it easy to write rules without regard for whatever else is already in the catalog, but may be inefficient. For instance, many guards may begin by testing against the same grammar. The flat organization would waste the result of those tests, repeating them over and over.

An organization that somehow indexes the rules by some properties of the guards is therefore recommended for actual implementation, but is not necessary for this discussion. Notice that the relative simplicity of the guards makes it possible to compare them syntactically, and thus facilitates automatic indexing. As a simple example, the user could write the rules in the flat model, and then have the grammar tests factored out automatically, this being trivial at least for non-recursive guards. This is an economy afforded by expressing the guards in a less-than-general language.

The experience reported in Chapter 6 shows that the flat organization does not introduce unbearable costs for the test suite described there, so no more mention of it is made in this dissertation, but it remains an interesting area for future research.

5.3.4. Rule Examples

This section shows a number of examples of rules, written in the language of SPOIL's catalog. Its purpose is to provide definiteness to the discussion above and a point of future reference for the example that follows, in Section 5.4.1.

In SPOIL, I represent rules by S-expressions. Symbols of the form **?var** represent unification variables. Before each rule is activated, the interpreter will assign the variables **?f** and **?c**, to the current fragment *id* and the current composition under investigation. Each rule is tested in turn: its guard is evaluated, and if successful, its transformation is run. The guard collects bindings for the variables that appear in the rules, and these bindings are made available to the transformation. Appendix A contains a list of the operations that may be used by the transformer.

Independent Sequence.

Guard.

```
(:and
 (:aspect ?f ?c ?a)
 (:graph ?a ?pg)
 (:parse ?pg "sequence" ?fparse)
 (:match ?fparse (:cc (:cc ?first ?tail) ?exit))
 (:part ?first ?a ?f.Seq.First)
 (:part ?tail  ?a ?f.Seq.Tail)
 (:flow-attribute ?f.Seq.First () :use ?Use.f.Seq.First)
 (:flow-attribute ?f.Seq.Tail  () :use ?Use.f.Seq.Tail)
 (:flow-attribute ?f.Seq.First () :def ?Def.f.Seq.First)
 (:flow-attribute ?f.Seq.Tail  () :def ?Def.f.Seq.Tail)
 (:intersection ?Def.f.Seq.First ?Use.f.Seq.Tail ?1)
 (:intersection ?Use.f.Seq.First ?Def.f.Seq.Tail ?2)
 (:intersection ?Def.f.Seq.First ?Def.f.Seq.Tail ?3)
 (:emptyset ?1)          ;not flow dependent
 (:emptyset ?2)          ;not anti dependent
 (:emptyset ?3)          ;not output dependent
)
```

Transformations.

```
(:apply
 (:add-to-execset ?f
  (:parallel
   (:execset ?f.Seq.First)
   (:execset ?f.Seq.Tail))))
```

This rule uses Bernstein's conditions to determine when two parts of a program are independent.

First we calculate the aspect of **?f** under collapse of the fragments whose *id*s are in **?c**. Then we extract the graph of the fragment **?f**, with the components already collapsed. That graph is then parsed against the SSFG grammar (the string "sequence" that appears in the parse primitive is just a comment[1]). Then we compare the parse with a pattern that represents two nodes in a row. The pattern can be read this way: A computation followed by a computation (:cc), where the first computation is another computation followed by a computation, whose first part is named **?first** and whose second part is named **?tail**. The second part of the outermost

[1] These comments are a historical artifact. An early version of SPOIL tried to deal with more than one grammar at a time; the slot currently taken by the comment was then reserved for a grammar reference. After that slot became useless (because SPOIL uses the SSFG grammar only), I left it in use because the comment helps to understand the rule.

:cc is named **?exit**.

Then the **:part** query tries to determine if the subgraphs **?first** and **?tail** so determined correspond to fragments. If that is the case, their flow attributes of usage and definition are computed, and three intersections taken: **?1** represents the definitions in First that are used in Tail, **?2** represents the definitions in Tail that are used in First, **?3** represents the definitions common to First and Tail. The guard is satisfied if the three intersections are empty.

When that happens, the transformer applies the transform. It just upgrades the execution set of the given fragment with the option of running the two parts discovered during the parse in parallel.

Forced Series.

Guard.

```
(:and
 (:aspect ?f ?c ?a)
 (:graph ?a ?pg)
 (:parse ?pg "sequence" ?fparse)
 (:match ?fparse (:cc (:cc ?first ?tail) ?exit))
 (:part ?first ?a ?f.Seq.First)
 (:part ?tail  ?a ?f.Seq.Tail)
 (:flow-attribute ?f.Seq.First () :use ?Use.f.Seq.First)
 (:flow-attribute ?f.Seq.Tail  () :use ?Use.f.Seq.Tail)
 (:flow-attribute ?f.Seq.First () :def ?Def.f.Seq.First)
 (:flow-attribute ?f.Seq.Tail  () :def ?Def.f.Seq.Tail)
 (:intersection ?Def.f.Seq.First ?Use.f.Seq.Tail ?1)
 (:intersection ?Use.f.Seq.First ?Def.f.Seq.Tail ?2)
 (:intersection ?Def.f.Seq.First ?Def.f.Seq.Tail ?3)
 (:or
  (:not (:emptyset ?1)) ;flow dependent
  (:not (:emptyset ?2)) ;anti dependent
  (:not (:emptyset ?3)) ;output dependent
  ))
```

Transformations.

```
(:apply
 (:add-to-execset ?f
  (:series
   (:execset ?f.Seq.First)
   (:execset ?f.Seq.Tail))))
```

The guard in this rule is essentially the negation of the previous one: instead of demanding that all the intersections be empty, this guard demands that at least one be not empty.

The transformation is also simple: it just adds to the execution set for **?f** the requirement of executing both parts strictly sequentially. (But, of course, later reorganizations and compensations could enable a parallelization that overrode this.)

Loop Recognition.

Guard.

```
(:and
 (:aspect ?f ?c ?a)
 (:graph ?a ?fpg)
 (:parse ?fpg "loop" ?fparse)
 (:or
  (:match ?fparse (:cc (:loop (:cd ?Body ?Test)) ?Exit))
  (:match ?fparse (:cc (:loop (:dc ?Test ?Body)) ?Exit)))
 (:part ?Body ?a ?LoopBody))
```

Transformations.

```
(:apply
 (:add-to-execset ?f
  (:sloop (:execset ?LoopBody))))
```

This rule locates the loops. It looks for parses that indicate a loop with the **?Test** either at the top or at the bottom. The SSFG grammar cannot generate arbitrary loops, but can handle large classes of almost-structured code, especially loops with multiple exits. So it could be used to support more general rules (with more branches in the **or** that inspects the results of the parse). The ones used above are sufficient for most purposes in languages like C, where loops are rarely implemented as nests of **if**s and **goto**s, but rather come from structured constructs.

Loop Distribution.

Guard.

```
(:and
 (:execset ?f ?e)
 (:match ?e (:sloop (:parallel . ?parts))))
```

Transformations.

```
(:apply
 (:map (:new-execset (:sloop ?part) ?newloop)
  ?part ?parts
  ?newloop ?newloops)
 (:combine-execsets :parallel ?newloops ?neweset)
 (:add-to-execset ?f ?neweset))
```

This rule can be explained as the following procedure:

(1) Determine if the current execution set is a serial loop of parallel parts.

(2) Iterate on the parts. For each of them, create a new execution set, a simple serial loop of that part. Combine all these execution sets into a parallel of all of them, then provide that new execution set as a choice for the execution set

of ?*f*.

5.4. Example

Now we go through the parallelization of the **`minmax`** example step by step. The flow graph and fragment structure of this program are repeated here for convenience.

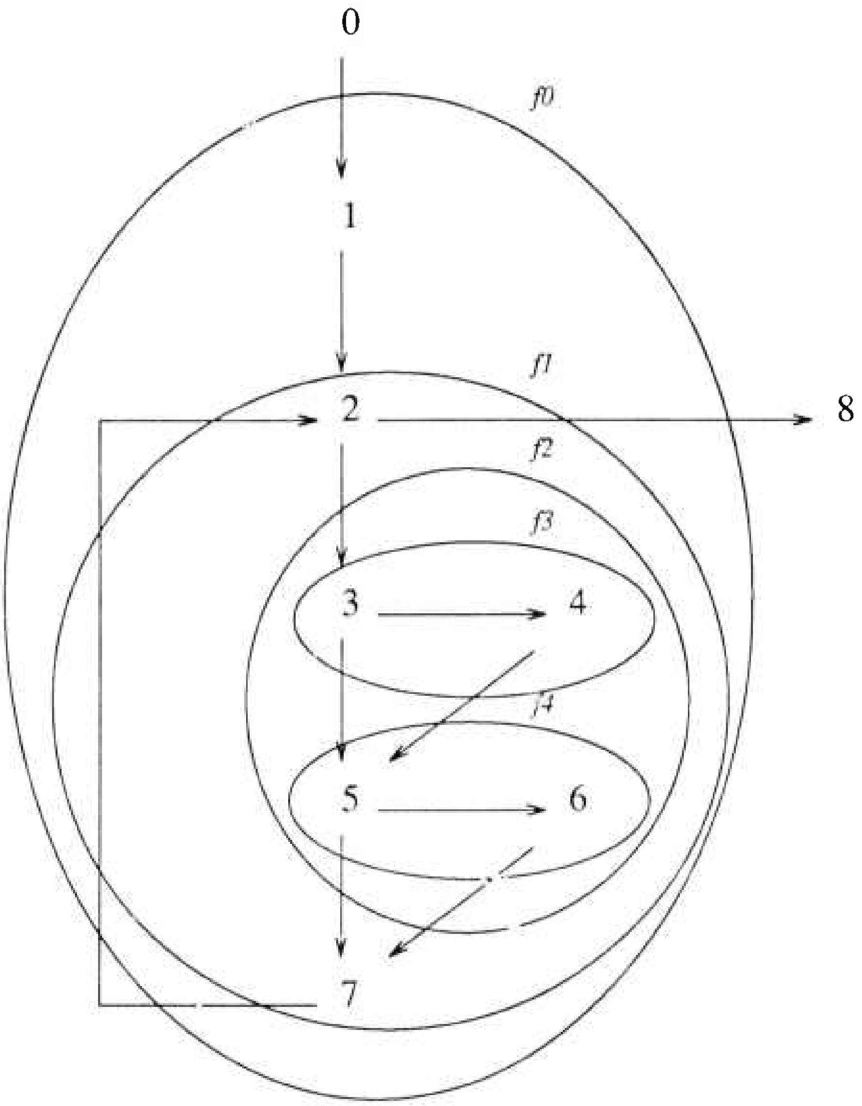


Figure 5.1. *dfi*-fragments for Minmax

We are going to carry out this example under the assumption that we are using a catalog that contains exactly the rules given as examples in Section 5.3.4.

The order of visit is calculated by obtaining the largest components of each fragment and computing the difference of the masses of the given fragment and each maximal component. Then we try to fit the masses of other subfragments in the residual.

One such order of visit is given below. The asterisks on the left indicate those fragments that are going to be discussed more carefully below.

```
        [1;8 / ()]
*       [2;8 / ()]
        [3;7 / ()]
        [3;5 / ()]
        [3;7 / (3;5 6;7)]
        [5;7 / ()]
*       [3;7 / (3;5 5;7)]
*       [2;8 / (3;7)]
        [1;2 / ()]
        [1;8 / (3;7)]
```

5.4.1. Move-by-Move Run

At the beginning of the transformation process, a new execset hierarchy is constructed. It is initialized trivially, so that fragment f_i has as initial execution set **(execset** f_i**)**. Then each of the compositions presented above is analyzed for possible parallelization. As we progress, some of the compositions match part of some of the guards, but fail on others. For instance:

[2;8/()]

This fragment corresponds to the loop of the example program, considered with *no simplification* of its internal structure. Although 2;8 is clearly a loop, it fails to satisfy the loop recognition guards, because its parse fails to have exactly one node standing for the body of the loop. The conditions, as seen above in the Loop Recognition rule, are:

```
(:or (:match ?fparse (:cc (:loop (:cd ?Body ?Test)) ?Exit))
     (:match ?fparse (:cc (:loop (:dc ?Test ?Body)) ?Exit)))
```

and, as can be seen, there is no binding for **?Body** when we consider the loop 2;8 with all its structure visible.

For various reasons, none of the intervening aspects completely satisfies the guard of any rule until we get to:

[3;7/(3;5 5;7)]

Here we find that the fragment in question (also named f_2, refer to Table 4.2) satisfies the S-grammar test. In addition, we can identify two sub-fragments (3;5=f_3 and 5;7=f_4) that are independent.

The grammar tests binds **?f.seq.first** to f_3, and **?f.seq.tail** to f_4. With these bindings, the action for this rule constructs a new alternative reading:

```
(parallel (execset f3)
          (execset f4))
```

From this alternative and the original trivial execset, the transformer will consider this new execset:

```
(choice
  (parallel (execset f3)
            (execset f4))
  (execset f2))
```

The reducer knows it is trying to provide a new execset for f_2, so upon recognizing it as a subform, it simplifies the execset to just the `(parallel ··· )` version.

[2;8/(3;7)]

This fragment begins with the execset `(execset f1)`. It matches the loop recognition rule, which provides a new reading as a serial loop. That reading changes the execset of f_1 to

```
(sloop (execset f2))
```

Now, the same fragment matches the loop distribution rule. The **?Parts** are recognized to be f_3 and f_4. With this information, a new execset is built:

```
(parallel (sloop (execset f3))
          (sloop (execset f4)))
```

The choice between these execsets cannot be simplified by the reducer. The former execset involves running the main loop serially, but using two parallel processes per iteration; while the new one implies a parallel of two sequential loops. These alternatives don't reduce to each other syntactically, as per the execset identities in §3.4, nor does one of them subsume the other unconditionally. In consequence, f_1 acquires the execset:

```
(choice
  (parallel (sloop (execset f3))
            (sloop (execset f4)))
  (sloop (execset f2)))
```

Please notice that the relative simplicity of the rules does not prevent them to match this aspect of the loop (in contrast to their failure with respect to [2;8/()]). The economy afforded by aspect formation is seen again in this simplicity of the loop rules, that are spared the need to match the unbounded variety of structure of general loop bodies.

When the last fragment has been parallelized, we have an execset hierarchy that assigns trivial execsets to all the fragments but for f_0, f_1, and f_2. Replacing these execsets recursively, the execution set for the loop appears as:

```
(choice
  (parallel
    (sloop
      (execset f3))
    (sloop
      (execset f4)))
  (sloop
    (parallel
      (execset f3)
      (execset f4))))
```

which expresses the sources of parallelism encoded in the little catalog that are also present in the program.

5.5. Execution Set Hierarchies

The final data structure relevant to the Transformer is the Execution Set Hierarchy. This structure keeps track of the association between fragment ids and their current (time-evolving) execution set expressions.

Initially, the association is trivial: with fragment *f*, we associate the execution set **(execset *f*)**. The Transformer modifies this association as explained above. Therefore, this structure eventually provides, for every fragment, its definitive execution set.

However, reaching that final execution set requires a certain attention to detail that complicates the intermediate stages. This section is devoted to explaining those details.

5.5.1. Implicit and expanded execution sets

Consider three fragments *f*, *g*, and *h*, progressively larger. At a certain stage in the transformation, the current value of their execution sets may be a function of their immediate descendants, for instance:

```
(execset h) = (choice ... (execset g) ...)
(execset g) = (sloop ... (execset f) ...)
```

Obviously, the execution set of *h* admits a different expression, namely **(choice ...(sloop ... (execset f) ...) ...)**. This expresses the execution set of *h* in ultimate terms: all the appearances of the form **(execset *x*)** refer to fragments *x* that cannot be expressed in any other way.

Given a fragment hierarchy *F*, its associated execution set hierarchy *E*, and a fragment *f* in *F*, I say that the *Expanded Execset Expression* for *f* is the execset expression that does not contain any **(execset *x*)** form for any fragment *x* that has a non-trivial execution set in *E*.

The expanded form of an execset expression relative to the hierarchies *F* and *E* can be obtained by recursively replacing the references to trivial execset expressions with their (then current) non-trivial counterparts.

An execset that isn't expanded with respect to the relevant hierarchies is said to be in an *implicit* form.

The possible difference between the implicit and the expanded forms is another redundancy in execution set representation, as I commented in Section 3.4.1.5. This redundancy is different, however, in that it can be shown only relative to current execution set and fragment hierarchies. In general, showing the equivalence of the expanded and implicit forms doesn't follow from the syntactic structure of the execsets involved alone, but depends on the structure of the program and on the history of previous successful parallelizations.

5.5.2. Propagating changes to execution sets

In general, the Transformer produces implicit execution sets. This can be seen by observing that the rules for adding new execution sets use as pieces the components of the current aspect. Those components are not required to have trivial execsets. However, matching execution sets that remain implicit is difficult. So there is incentive for keeping both the implicit and the expanded forms. As a matter of implementation strategy, then, we keep both forms at all times. At the end of the parallelization, we need only the expanded one. So, when do we need the implicit forms?

Well, we need the implicit forms only in the intermediate stages. They keep the connection between fragments and their subfragments that have triggered parallelization. In the example above, the form **`(choice ... (execset g) ...)`** remembers that g and h are related. If we replace it immediately by the expanded form, we would run the risk of losing later opportunities for parallelization.

This would happen as follows: The eventual enumeration order guarantees that the Transformer will visit large after small, but nothing more. So it is perfectly possible for us to visit f, then g, when parallelizing h. If we replace at the first success the execution set for h with its expanded form (in terms of f), all traces of the connection with g will be lost. If we later succeed in parallelizing g, the eventual enumeration order guarantees that we still have a chance to look at h again. Regrettably, that new attempt couldn't benefit from the intermediate parallelization, because the appearance of g in the execset expression for h has been lost.

This is the reason for keeping both forms of the execution set at all times. The implicit version remembers the connection between a fragment and the aspects under which it was parallelized; the expanded version shows the ultimate effect of all the parallelizations. To compare execsets, the guard evaluator refers to their expanded forms; to update execsets, the transform evaluator modifies first the implicit form, but this modification isn't enough to maintain the correctness of the hierarchy, so the transform evaluator also updates the expanded form of the fragment just parallelized, and then re-expands the implicit forms of all the ancestors of that fragment, starting by the nearest ancestor and going upwards in the hierarchy. Updating the execset expressions in this order is necessary to guarantee that the effect of the transformation propagates correctly to all the fragments that could be still parallelized by using this information.

To summarize, we start with trivial execution sets as both the implicit and the expanded forms of every fragment. When a transformation occurs, the implicit execset is the one directly returned by the Transformer. That execset expresses the transformation in terms of the aspect that was involved. We add this execution set to the choices kept in the implicit form, and reduce it. Then we expand the reduced form with respect to the current hierarchies, and remember that expanded form along with the implicit one. Finally, we climb the hierarchies, re-expanding the current implicit forms of all the ancestors of the modified fragment.

5.6. Discussion

A few observations are still desirable. First of all, we can judge the complexity of a parallelizer by inspecting the computational power of the rule set used.

The reason for specifying the primitive guard predicates, and for limiting the guards to using them, is precisely to have a chance to observe the complexity of testing for parallelism in a single place, rather than spread all over the compiler. The rule set discussed depends essentially on linear unification, set operations, and graph parsing. Arithmetic and other universal modes of computation were intentionally not used, to avoid adding power that wasn't absolutely needed. Of course, set operations can be used to bring in arithmetic, but the rules would become impossibly long then. The interest was in minimizing the temptation to implement the parallelizations via arbitrary code within the rules (then a single rule could conceivably do all the work!).

The conclusion of this study was to confirm the impression that the complexity of parallelization can be reduced to the complexities of data and control flow analysis, and of graph parsing, glued together by unification of S-expressions. It remains an open question to characterize the minimal power needed by those components. Chapter 6 shows a possible arrangement with respect to graph parsing, but no claim is advanced as to its minimality.

Another formalization of interest is that of relative completeness of a parallelizer. The general case of parallelization is undecidable, on account of the undecidability of certain data flow problems [Ber66], so we cannot expect to obtain an algorithm that finds *all* the conceivable parallelism in arbitrary ordinary programs. However, the exhaustive search procedure used in my Transformer affords me a *relative* sense of completeness: my Transformer finds any parallelization that has been expressed in its rule catalog and whose guard can match in the program under consideration, perhaps after other transformations have exposed a structure of interest.

To see why, observe that the eventual enumeration order ensures that synthesized parallelizations (i.e., those made possible only after recursive consideration of the sub-fragments) are successful when at all possible. As for inherited parallelizations (i.e., parallelizations made possible by inspecting a super-fragment), I claim that the hierarchical nature of the fragment structure ensures at least that one can write the rule to match the structure of interest in the context of the larger fragment.

These issues deserve a more detailed formal analysis. The ideal setting for that analysis is in general program transformation theory, and so it may pay to embed the question of relative completeness (of a Transformer with respect to a Catalog) in the question of relative completeness of an arbitrary program transformation scheme with respect to non-deterministic rewriting systems. (The non-determinism may be needed to account for having to choose which aspect actually exposes the structure that permits parallelization.) Accomplishing this would afford us a declarative parallelizer.

5.7. Related Work

The idea of collecting transformations that could improve the performance of generated code seems to have started with Allen and Cocke's catalog [AlC72].

Parafrase isolated transformations in specific modules, and insisted on a source-to-source mode of operation. Thereby, one could experiment with reordering (or repeating) transformations. Each of the independent modules could be considered a member of a loose catalog of general transformations.

Aiken [Aik88] shows a parallelizer that uses a small set of rules (four) to accomplish its goals. His catalog is tightly woven with the control structure of the compiler.

I want to point out that the literature often lumps together transformations of dissimilar natures. I will argue here that the clear distinctions introduced by my model clarify the proper role of those transformations.

For instance, consider the transformations discussed by Polychronopoulos in [Pol88]. He distinguishes some common optimizations, like induction variable substitution and loop unrolling, that apply generally to all optimizers. These transformations do not necessarily expose parallelism by themselves, but prepare the program for future transformation. Then he discusses (Section 2.3) transformations that are ostensibly specific to vectorization/parallelization.

Among those transformations, he discusses loop interchange (Section 2.3.2), loop vectorization (Section 2.3.1), and loop blocking (Section 2.3.5). Observe that the first one converts the sequential program into another sequential program (thus belonging properly into the front end of my design); the second is a real parallelization, and of rights belongs in the catalog of my model; and the third one actually seems to hide some parallelism!

To see why, let's review what *loop blocking* is. It reduces to converting a single loop into two nested loops, such that the inner loop proceeds in chunks of uniform size. For instance, adapting the example in [Pol88, p. 26], we get from:

```
for (i = 0; i < N; i++)
    a[i] = b[i] + c[i];
```

the new program:

```
for (j = 0; j < N; j += K)
    for (i = j; j < MIN(j+K, N), i++)
        a[i] = b[i] + c[i];
```

The motivation for this transformation is, presumably, to adapt to architectural characteristics (like the number of vector registers). Both of the loops above have essentially the same execution set: `(ploop (execset` *f*`))`, where *f* is the name of the fragment that contains exactly the assignment statement.

Once one adopts the model discussed in this work, the role of loop blocking is clear: it is actually a back end transformation, most certainly not necessary in the parallelization catalog.

We will see later that SPOIL's catalog can represent front end and back end transformations, in addition to the transformations that properly belong there (i.e., those transformations that enlarge execution sets). This flexibility is useful to the implementor of specific parallelizers, but there is now a characterization that will recommend where to put specific transformations. One can use that characterization as a criterion to decide between different configurations.

CHAPTER 6

SPOIL: Practical Experience

This chapter presents the experience gained through SPOIL. First I describe the implementation. Then I consider the observed costs of various design decisions. These costs can be used to balance the choices available to the designer, namely the grain of the analysis and the thoroughness of the search among the possible transformations, versus the running time of the compilation. Finally, I discuss the coverage of the transformation space explored by the catalogs used in these experiments.

The experiments described here have several purposes. The first is to validate the attempt to separate parallelism detection from the other chores of the parallelizer. A second purpose is to test the specific combination of grain and composition strategy used in SPOIL's fragment hierarchy. A third objective is to collect and validate a catalog of modest size, yet capable of encoding useful transformations.

6.1. The Input to SPOIL

This section describes the language accepted by SPOIL, and discusses the consequences of some of the decisions that derive from that choice of input.

SPOIL processes a subset of C [KeR89]. This subset excludes the notion of volatility, and is restricted to one function per unit of compilation (i.e., per `.c` file)[1].

The choice of a subset of C was suggested by a number of technical conveniences, not the least being the availability of the GNU C compiler [Sta89]. It provides convenient access to intermediate representations, which I used to generate the input expected by SPOIL, as described below in Section 6.2.

C is widely used, and so should be an interesting target for parallelization. However, it has not received attention proportional to its popularity, probably because it has been mainly used as a systems programming language, an area of application not traditionally concerned with parallel speedup. As C spreads its domain to application programming one can expect that interest in its parallelization will increase.

One previous parallelizer for C is due to Allen and Johnson [AlJ88], who have worked on compiling C for a multiple SIMD machine (Ardent's Titan). Their insight was to raise the semantic level of their intermediate representation, in order to make

[1] These limitations are not inherent in the general model discussed in this dissertation, but are conveniences to the prototype implementation. When discussing SPOIL it is important to distinguish those features, advantages or disadvantages, that are induced by the theory discussed here, from those other features that are just accidents of the prototype implementation.

the dependences stand out. The abstraction provided by my fragment hierarchy has a similar purpose, but differs in that it imposes no details on the actual intermediate representation used, while Allen and Johnson describe a specific intermediate representation.

SPOIL relies on a front end to parse and check the semantics of the input program, and accepts as its input proper an intermediate representation of the program, where all syntactic characteristics of C have been eliminated.

6.2. Implementation Considerations

SPOIL is written in Common Lisp [Ste84]. It was developed and tested using KCl (Kyoto Common Lisp, [YuH85]) and KCl's AKCL port.

SPOIL uses GCC as heavily as possible to avoid having to deal with problems solved by existing technology, namely: lexical and syntactic analysis, type checking, and conversion into a register-transfer-level (RTL) representation [DaF84]. SPOIL reads in the RTL representation of the program, collects the instructions into granules, and computes granule-level data flow information.

The proof-of-concept nature of SPOIL dictated a number of restrictions on the format of the programs used for the experiments, the most important being the limitation to one function per compilation. Interprocedural analysis, though an important issue in itself and in its possible applications to parallelization, would not add much weight to the test of the ideas exposed in this dissertation, so no effort was made to parallelize across procedure calls. To accommodate interprocedural analysis in my model, one needs to define fragment hierarchies that span the call graph, and extend the dataflow information across procedure boundaries (see [Cal88, CoK88, HRB88, Li88] for examples of the analyses that would be needed to extend my model).

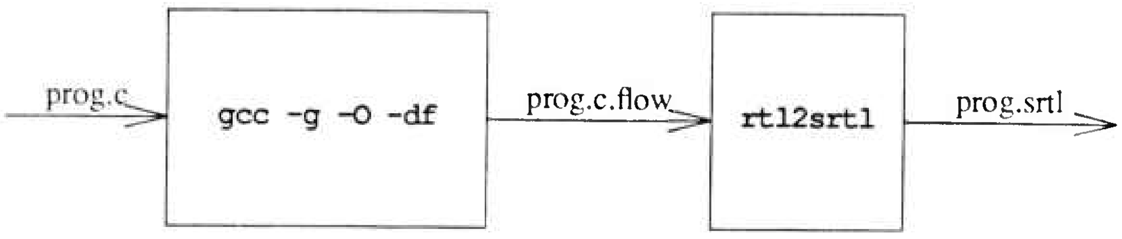


Figure 6.1. From C to SRTL

Figure 6.1 shows schematically the way a C program is converted into the input for SPOIL. Given a C function in a file **`prog.c`**, we run GCC on that file, producing files **`prog.c.flow`**, **`prog.c.greg`**, and **`prog.s`**. The first two contain alternative views of an intermediate representation for the program, the last one contains assembly code. The intermediate representation (*RTL*) is described in [Sta89]. (See also [DaF84].)

The RTL output is not directly readable by SPOIL. The program **`rtl2srtl`**, reproduced below in Figure 6.2, cleans it up, producing a file **`prog.srtl`**, which is suitable as input for SPOIL.

```
#!/bin/sh

echo "( ;; RTX list"

# Eliminate header.
sed \
    -e '/^[^(]/d' \
    -e '/^(/{
            :loop
                n; b loop
            }'

# Eliminate flags and machine modes.
sed \
    -e 's/\([^:/][^:/]*\)[:/][^ ]*/\1/g'

# RTL vectors are written [...], convert to #(...)
sed \
    -e 's/\[/ \#\(/g' \
    -e 's/\]/\)/g'

# Squeeze empty lines
cat -s

echo ")"
```

Figure 6.2. From GCC's output to SPOIL's input

This redundant representation of the program (three files coming out of one source) was needed only to extract enough information to map temporary register names into user variables, which is a convenience when debugging this code, but not mandatory for this work.

The activity of SPOIL proper begins with the `.srtl` file. As described in Chapter 4, the list of instructions is successively converted into a list of granules, a list of fragments, and a fragment hierarchy.

An initial execution set hierarchy is also constructed. For every fragment *f* in the program, that initial execution set hierarchy contains either the execution set denoted by **(execset *f*)** or the one denoted by **(sloop (execset *f*))** depending on whether *f* is a trivial loop.

Some comment is needed about these initial contents. Given that the execution set notation allows one to express directly the looping structure of a sequential program, one could expect to find all the loops designated in the initial hierarchy with a **sloop** construct. After all, determining the loop structure from the control flow graph is a well-known problem [ASU86, Chapter 9].

On the other hand, that determination is nothing more than the test for a very conspicuous flow pattern, namely the presence of a backwards arc. Finding such patterns is the domain of the SSFG graph parser.

This means that SPOIL could have chosen to identify the (serial) looping structure either in the front end (during standard control flow analysis) or with the transformer. I decided to leave the loop detection rules in the catalog, but out of convenience identified the trivial self loops (i.e., those granules that branch to themselves) during control flow graph construction. This is an example of a possible tradeoff between doing something in the front end and doing it in the parallelizer. The existence of such tradeoffs illustrates the flexibility of this model to accommodate efficiency considerations.

All this preparation leads to the application of the transformation algorithm described in Chapter 5. SPOIL's work finishes with the production of a definitive execution set hierarchy for the program (including, perhaps, new code that was added in the course of exposing parallelism) and the output of some run-time statistics.

There are actually two versions of SPOIL. In one of them (which I will refer to as the **bblock** version), the granules are basic blocks of the RTL code. In the other one (the **cstat** version), the granules correspond roughly to statement boundaries in the original C source. Therefore, the **cstat** version operates at a finer granularity. The **bblock** version was used originally to test the general soundness of the approach and to experiment with mid- to coarse-level parallelizations. The **cstat** version has been used to reproduce vectorization results.

6.3. Costs, Tradeoffs

The conceptual clarity obtained by separating the detection of parallelism from its exploitation, and by encoding parallelization knowledge outside the parallelizer itself, is gained at a price. The enumerator must generate a number of subsets of the program, and the transformer must test all (and transform some) of them, ostensibly without benefit of architectural constraints.

Naturally, this opens up opportunities for tradeoff. A fragment hierarchy with small fragments is going to explore many more possibilities than would one with relatively large grain (like the one used by the **bblock** version of SPOIL), but can find more parallelism. On the other hand, the more sub-fragments a fragment has, the more aspects will have to be inspected, which increases the total running time.

First, let's consider the question of size of the fragment hierarchy used by SPOIL. As already explained in Chapter 4, the fragments used by SPOIL are those single-entry/single-exit regions of the control flow graph that also happen to be contiguous in a depth-first numbering of the basic blocks, which I called *dfi-fragments* in Chapter 4 (see Section 4.1).

For general graphs, there could be as many as $O(B^2)$ such dfi-fragments, where B is the number of nodes in the graph (i.e., basic blocks or C statements in the program). On the other hand, control flow graphs are observed to be a very restricted kind of directed graph. None of their nodes has out-degree greater than two. This constant bound makes it likely that the number of fragments in actual programs will be more linear than quadratic.

To test this hypothesis, a suite of programs was gathered in order to measure the ratios of fragments and aspects to granules, and to test the overall coverage of known

Name	B	F	A
wolfe-4.2	6	4	7
s131	7	5	9
s123	7	4	7
s124	8	4	7
minmax	9	6	10
minmaxsum	11	9	16
s115	11	8	15
s2710	12	5	9
sploops	14	9	16
solve	16	14	29
frac	21	11	20
steele	37	47	100
decomp	78	90	201

Table 6.1. Medium Grained Test Programs

parallelization techniques. The programs are included in the Appendix B.

Table 6.1 and Figure 6.3 show the results for the **bblock** version. A total of 13 programs were considered, ranging in size from 6 to 78 basic blocks. In that table, B stands for the number of basic blocks, F for the number of fragments corresponding to those blocks, and A for the number of aspects considered for transformation.

A linear least squares fit to the data in Table 6.1 shows the relation

$$F = -6.01 + 1.24 \times B \quad (\rho=0.98) \tag{6.1}$$

This result supports the assumption that fragment count doesn't grow faster than linearly with the size of the program, as the measures are consistent with an $O(B)$ growth rate.

Even given a linear number of fragments, we can still observe large fragment hierarchies if there are too many aspects. This is possible due to the potential existence of many maximal components for the larger fragments (for instance, for the entire program). Choosing a strategy for aspect selection is thus critical to the efficiency of this class of parallelizer.

The test suite behaves modestly with respect to aspect formation too. The linear least squares fit (also from Table 6.1) shows the relation

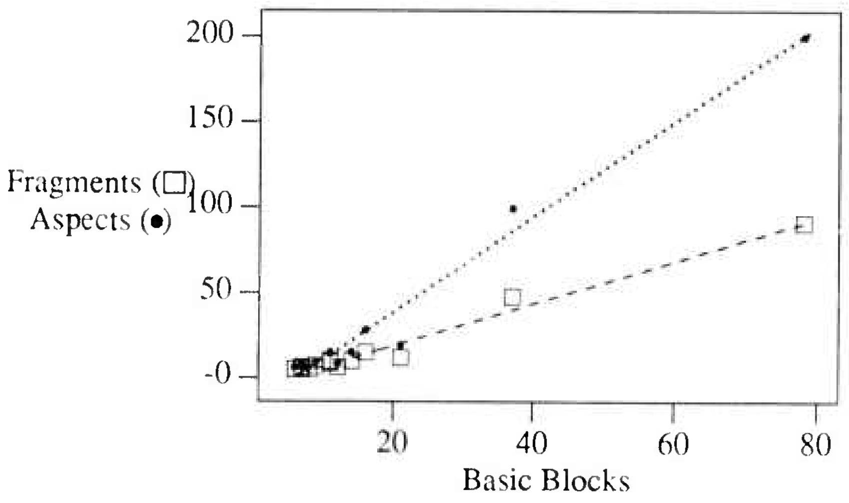


Figure 6.3. Medium-Grain Experiments: Sizes

$$A = -16.6 + 2.79 \times B \quad (\rho{=}0.98) \tag{6.2}$$

where again B is the number of blocks and A is the number of aspects.

These correlations are strong enough to support experimentally the claim that both the fragment and the aspect counts grow only linearly with the size of the program (measured in basic blocks).

Similar results hold for the **cstat** version of the experiments. The sizes are given in Table 6.2. The relations (represented also in Figure 6.4) are now:

$$F = -12.82 + 3.44 \times G \quad (\rho{=}0.99) \tag{6.3}$$

and

$$A = -58.74 + 8.26 \times G \quad (\rho{=}0.99) \tag{6.4}$$

where F and A designate fragment and aspect counts, respectively, and G represents granules (C statements).

6.3.1. Discussion

Having linear amounts (on granule count) of fragments and aspects is necessary for efficiency in this approach. As aspect formation could potentially construct exponential numbers of aspects, it is reassuring that real programs don't exhibit an explosive behavior.

Indeed, although the linearity of fragment count on granule count was more or less expected from the constant out-degree of control flow graphs, there was no similarly strong hint that aspect count would remain linear too. An important open question is then the characterization of fragment and aspect definitions that ensure modest growth rates with respect to granule count.

Name	G	F	A
loop1	9	13	23
loop2	9	12	20
loop3	9	12	20
doacr	10	16	29
pat	14	33	70
frac	37	130	139
solve	39	139	329
decomp	134	441	1056

Table 6.2. Fine Grained Test Programs

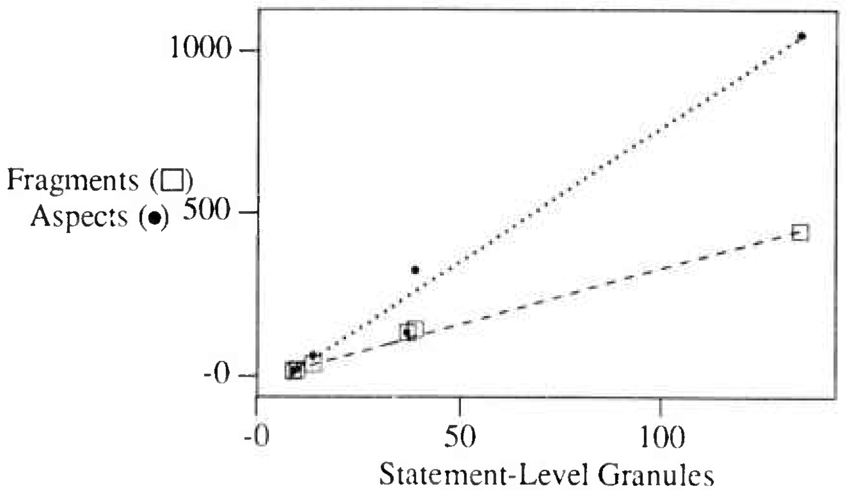


Figure 6.4. Fine-Grain Experiments: Sizes

Still, the relations desciibed above don't imply that the total cost of fragment hierarchy formation will be linear in the size of the program. SPOIL currently generates fragments by enumerating $O(G^2)$ pairs of integers, back-mapping from the depth-first numbering of the granules, and then testing for dfi-fragmenthood. This generate-and-test approach was acceptable for the proof of concept implementation; but knowing now that there is only a linear number of fragments to be found suggests exploring a more discriminating enumeration of depth-first intervals, perhaps by extending known ones incrementally, instead of enumerating all the possibilities and discarding the failed ones. The question of whether the total cost can be reduced from quadratic is still open.

Also, the programs used in these experiments have been quite small (although some of them, like **solve** and **decomp** are fair representatives of typical numerical kernels). It remains to test this approach against much larger programs, to rule out the possibility of higher-order effects expressed at those larger sizes.

Although actual running times don't provide a sufficiently firm basis for comparison, it may be useful to mention that running SPOIL doesn't cost excessively in real time. The prototype was instrumented to report running times at two opportunities: when the main data structures had been prepared (first fragment and execution set hierarchies, original visit order), and after the transformer had finished running. The first time included time spent outside the Lisp system, running GCC twice, and running the scripts that massaged GCC's output. The trial runs took (on a Sun SPARCstation 1, the two terms being the two times mentioned above) between

3.8+7.5 seconds for **minmax** and 222.3+241.0 seconds for **decomp**. Although not industrial-quality yet, these times are still tolerable in a prototype, and show that there is reason to attempt a new implementation focused on performance.

6.4. Coverage of Transformations

The previous sections have shown that the approach taken by SPOIL is tolerably efficient. It remains to be seen how generally applicable it is. To this end, I will compare here SPOIL's catalog to already known parallelization results, in order to support the claim that separating detection from exploitation of parallelism does not hinder the power of the parallelizer.

The original version of SPOIL (the **bblock** version) operates at a rather coarse level of granularity. The literature has generally considered fine-grain parallelizations (vectorization, in particular), so to compare SPOIL to classical results one needs to refine its grain. In Section 5.4 I have shown an example of the operation of SPOIL on medium-grain parallelizations.

The plan for the rest of this section is as follows: first I compare SPOIL's catalog with classical transformations (vectorizations). The comparison is done by reproducing the previous results in the language of my catalog. Then I consider an especially hard case, where the comparison is against a prior result that depended strongly on *not* separating detection and exploitation. Then I consider the possible encoding of the fairly general class of transformations that introduce synchronization to turn a serial loop into a parallel loop. Finally I consider another special case, where both coarse and fine grain parallelizations are possible, and where SPOIL can find both uniformly.

The transformations used in this comparison essentially exhaust the ones found in a careful analysis of the literature. Their different origins and independent development made it hard to organize these comparisons more systematically, so the following sections can give the reader the impression that these techniques are only haphazardly related. This feeling of unsystematic enumeration of techniques was part of my motivation for undertaking this study. It was with great satisfaction that I saw them fit the framework developed in this work, thus putting them in a context where their common traits are clear.

6.5. The Classical Transformations

To test SPOIL's competence against standard transformations, I used the programs `loop1`, `loop2`, `loop3`, `steele`, and `sploops`, which can be found in the Appendix B. The rules provided in the catalog (appendix A) were sufficient to produce the transformations I describe below.

In `loop1`, the two statements $S1$ and $S2$ happen to be independent. The rule for Loop Distribution converts the serial loop into a series of two `ploops`, thereby catching the vectorization with a grain-independent rule. This is one of the advantages of the approach advocated here: the detector finds the opportunity for parallelization, and makes it available to the back end without either (1) prejudicing whether the series of `ploops` is the desirable thing to do, or (2) (assuming it is) whether it will be implemented as two tasks, or as multiple passes through a vector engine.

loop2 and **loop3** show the use for the rules of Partial Distribution. Here we ask the data flow analyzer to tell us if the chosen decomposition of the current fragment has a part that is loop-independent (that is, its dependences form a strongly connected region and don't go across iterations). In such case, the strongly connected region can be extracted [Pol88, Section 2.3.1] and parallelized apart from the rest.

Notice that the proposed rules don't work if the separable part happens to be inside the loop, surrounded by non-separable parts. To have that work we need to ask of the front end that loop bodies be presented in topological sorted order by their dependences, or to improve the matcher (by matching on the dependence graph, not on the control flow graph). This limitation has not affected the results on my test suite, but it should be considered for future work.

6.6. Forcing The Separation: Ignorable Self Anti-Dependences

Wolfe [Wol89, Section 3.4] presents an interesting case. He considers the possibility of just ignoring self anti-dependence cycles by using knowledge about sequencing guaranteed by the architecture. For instance, in:

```
/* adapted from [Wol89, Section 3.4] */
for (i = 0; i < N; i++)
      a[i] = a[i+1] - 1;
```

one finds the following behavior: in the first iteration, `a[0]` gets assigned `a[1]`, which itself gets modified in the next iteration by `a[2]`, etc.

So, in appearance, we cannot reorganize the loop for parallelization, due to the ''clocked'' behavior of the moves to `a`.

However, if we knew that **N** is less than the number of vector registers in a machine that guarantees to perform all the vector loads before any of the stores, then it would seem we could go ahead and parallelize the loop in spite of the obvious dependences. Of course, if we don't know the value of N or its relation to the vector width, we can always do loop blocking.

The challenge to the method advocated here is that the success of the vectorization depends on (implicit) guarantees on the order of loads and stores.

Can we do the same without breaking the separation between detection and exploitation of parallelism?

It turns out that it is indeed possible. Consider the rule for ignorable self-antidependence cycles from appendix A, repeated here for convenient reference:

```
(catalog.add-rule                              ;wolfe p.64
 cat
 "Ignorable self-antidependence cycles"
 (guard.new '(:and
              (:exists-execset ?f (:sloop (:execset ?fb)))
              (:flow-relation ?f ?fb ?fb :flow-dependent)
              (:not (:flow-relation ?f ?fb ?fb :anti-dependent))
```

```
               (:not (:flow-relation ?f ?fb ?fb :output-dependent))
               (:flow-attribute ?fb () ?def-locs ?d)
               (:match ?d (?target))
               (:code ?fb ?oldcode)))
 (transform.new '(:apply
                   (:new-name ?new-array)
                   (:new-code ((call-insn 'copy ?target ?new-array))
                    nil ?copy-in)
                   (:with-new-name ?target ?new-array ?oldcode ?newcode)
                   (:new-code ?newcode ?newbody)
                   (:new-code ((call-insn 'copy ?new-array ?target))
                    nil ?copy-out)
                   (:new-execset (:series
                                  ?copy-in
                                  (:ploop ?newbody)
                                  ?copy-out)
                    ?new-eset)
                   (:add-to-execset ?f ?new-eset))))
```

What this rule does is to convert the implicit guarantee of serialization into an explicit one, by introducing a temporary array to hold the new left-hand sides while the old ones are still being used. The back end can now be informed (perhaps by introducing a storage class to this effect) of the caching nature of the temporary that the front end has introduced. If the back end maps the temporary into the vector registers, it would have eliminated the actual copies and allocation, and *it* can then exploit knowledge about the machine. On the other hand, if no such guarantees are given by the hardware, but the copies happen to be cheap due to any other reason, the back end has available a locus of parallelism that could be useful, but that in no opportunity is incorrect (because its conditions for correctness are enforced explicitly).

6.7. DOACROSS – A General Class of Loop Parallelization

Here I consider a very general class of transformations that add synchronization operations to serial loops in order to run them in parallel, yet honor the dependences. The reference here is program **doacr**, from [Wol89, p. 76]. The critical points are these:

(1) We can (at least in the **cstat** version) query for the iteration distances between statements in the loop. Conceptually, we can ask the data flow analyzer to provide us with the intermediate code list needed to replace the original one to secure synchronization. Producing and altering intermediate code lists is in the domain of the front end so this requires no special purpose mechanisms other than an interface to request these lists when needed. This view is expressed in the following rule:

```
(catalog.add-rule                        ;wolfe/doacr
 cat
```

```
        "DOACROSS"
        (guard.new '(:and
                     (:exists-execset ?f
                                      (:sloop . ?loopbody))
                     (:match ?c (?body))
                     (:exists-execset ?body
                                      (:series . ?loopbody))
                     (:flow-attribute ?f (?body)
                      :synchronized-code ?synch-code)))
        (transform.new '(:apply
                         (:map (:apply
                                (:match ?synch (?fid ?newcode))
                                (:new-code ?newcode ?fid ?nfid)
                                (:new-execset (:execset ?nfid) ?enfid))
                          ?synch ?synch-code ?enfid ?enfids)
                         (:combine-execsets :ploop ?enfids ?new-eset)
                         (:add-to-execset ?f ?new-eset))))
```

This rule essentially depends on there being a data flow analyzer for the attribute **:synchronized-code**. This is consistent with our previous observation that some transformations, even when doable in the parallelizer (we could have asked for the iteration distances ourselves, and depended on the back end to eliminate redundant waits and posts), should be actually moved to the front end (because they are easier there) or to the back end (because they are architecture-dependent).

6.8. Operating Uniformly at More Than One Grain

I observed above (while discussing the **loop1** example), that some rules can capture the available parallelism of a fragment very cleanly, independently of the granularity of the tasks in the target machine. In **loop1** it turned out that the same rule discovered what can be called either a concurrentization or a vectorization (depending on whether one can add two vectors into a third, a commonly available operation) with just one test. A similar thing can be observed in the **pat** example (reproduced below):

```
/* Adapted from Smith and Appelbe, ICCP 1988,
 * ``PAT -- An Interactive Fortran Parallelizing Assistant Tool''
 */

int
main()
{
    float           a[20], b[20], sum, max;
    int             i, j;
    extern void     init();

    init();
    sum = 0;
    j   = 20;

    for (i = 0; i < 20; i++) {
        sum = sum + a[i];
        if (i > 0)
            a[i] = b[j] + a[i - 1];
        if (b[i] > max)
            max = b[i];
        j = j - 1;
    }

    printf("sum : %f8.5 max : %f8.50, sum, max);
}
```

This program is a C version of the **pat.f** program (both in appendix B). The original appeared in [SmA88, Diagram 1]. The parallelization of interest here introduces explicit synchronization in the loop body. Notice that this program shares with the **minmax** example the possibility of distributing the sum and the max computation, and to partially distribute the vector shifted addition in the middle. SPOIL's catalog would apply *all* of these transformations, and present them as options to the back end, while PAT apparently concentrates on the coarser-level transformations. It is interesting to notice that, while SPOIL provides choices to the back end, PAT is designed to provide interactive choices to the user. One could imagine a merge of the two approaches.

6.9. Discussion

My experience with SPOIL has shown that suitable definitions of granule, fragment, and aspect permit a successful implementation of the already known parallelizations.

With respect to practical applicability, the most important feature was the selection of aspects. One of my early tries involved a medium grain analysis (granules were basic blocks) and a rich fragment hierarchy. Aspects were formed by fitting any subfragments to the residual masses after one collapse, thereby matching every maximal component with all the components (no matter how small) disjoint to it. The hope was that there would be not too many aspects after all.

Soon it was clear that that wasn't the case. Although very small programs seemed to show a linear cost, there were a few examples (like the **decomp** program) that required seemingly exponential numbers of aspects. This happens whenever the entire program contains many ''small'' fragments.

Two possible cures were tried. The first was to avoid all the single-granule fragments, except for the self-loops. Although this improved conditions (and can be defended in terms of quality of abstraction: single-granule fragments don't abstract any structure), it wasn't sufficient to yield a practical fragment hierarchy.

The second attempted cure was to try only the maximal components when decomposing a fragment. The combination of this and the first approach yields the linear-size hierarchies discussed above. But this severe pruning of the possible aspects could hide useful parallelizations. What the examples above show is that that is happily not the case. So, it is possible to enjoy the intellectual leverage obtained by performing the recommended separation of concerns, while not renouncing any of the generally useful parallelizations.

These experiments have established my confidence in the applicability of the thesis of this work. SPOIL has been tested successfully, both in terms of coverage of previous work and of indication of practical efficiency, without so far showing any serious flaw in the model (as opposed to shortcomings in the prototype). Within its proof-of-concept scope, SPOIL supports the claims of this dissertation.

CHAPTER 7

Conclusions

7.1. Summary

It is time to look back at the claims put forth in this work, and to summarize what has been learned in the process.

The object of this study has been the parallelization of ordinary code. By such, I mean imperative code with sequential semantics, further constrained by the need to represent accurately the control and data flow relationships. I have claimed that the total task of parallelization is intellectually more manageable if one separates the concerns of detection and exploitation of parallelism. This investigation has concentrated on the organization of the detection component, aiming to support the larger claim about parallelization in general.

It is easy to believe that keeping apart two demanding purposes, as detection and exploitation of parallelism are, has to result in somewhat cleaner approaches to each of them. What is not immediately clear is whether this separation is possible, or whether it will introduce problems harder than those solved by its means.

I believe that the preceding pages have shown how indeed it is possible to keep the two concerns separate. The main insight is to make sure that a explicitly parallel representation (like the one provided by the execution set expressions) be used as the interface between the two problems. To recapitulate, the need for explicit parallelism in the interface is a reflection of the need of the back end for positive information about parallelism. The standard dependence representations are good at telling us what *can't* be done in parallel. It is not immediately obvious how one goes from that form of negative information to figuring out what *can* be done in parallel.

So, I have described a general schema of program transformation, that yields an explicitly parallel representation.

This schema (the highest level of abstraction discussed in the introduction, Section 1.3) can be characterized by two properties.

The first is a hierarchical description of the program, which provides us with the means of abstraction needed to eliminate extraneous features from consideration, while permitting us to arrive at the parallelization of large, possibly complicated, pieces of code by composing the results of the analysis of their subparts.

The second is another separation of concerns. The model distinguishes the control structure used to enumerate parts of the hierarchy from the specific transformations that expose larger execution sets. This separation permits us to capture the observation that 'perfect' parallelization is generally undecidable (by reduction from halting [Ber66]), while being able to reason that a given parallelizer, working within polynomial resource bounds, won't miss any parallelization it 'should' find.

The experimental evidence accumulated via SPOIL confirms the applicability of this approach.

So, when wanting to write a specific parallelizer, I would proceed this way (and this dissertation argues that so should *you*):

(1) First of all, I would figure out a unit of resolution for the analysis. Suitable candidates are individual register transfers, statements in the language (at least, assignments), or basic blocks. These are the *granules* of the analysis. This unit of resolution imposes a bound on how thorough will be the search for parallelism, but simultaneously provides a way to control the resources spent in that search. Granules are also the unit of parallel execution. The number of granules in a program will be considered its size. A constraint on our freedom to define granules is that there should be a practical way to describe data and control flow relationships among granules.

(2) Second, I would consider a hierarchical structure on program parts. At the lowest level of the hierarchy we have some or all of the granules. Then, at higher levels, we have control flow structures composed of elements of the lower levels. Call the elements of this hierarchy *fragments*. Two conditions restrain our freedom to define fragments: (1) the cost of going from granules to fragments should be bound by a polynomial on the size of the program, and (2) there should be a convenient way to infer data and control flow relationships among fragments from the relationships known to hold among granules. (Most of the elaboration so far has counterparts in previous parallelization work, the contribution here being having put them in a common framework sufficiently generic to encompass the previous approaches.)

(3) Third, I would consider "quotients" of fragments with respect to sets of their subsets. By a "quotient" here I mean the collapse of the internal structure of the named subsets, such that they appear as single nodes for the purposes of this analysis. A fragment, considered along with the collapse of some of its subfragments, is what I call an *aspect*. (The notions of aspect and of driving the analysis by aspects, are, as far as I know, a novelty of this work.) Of course, the same restriction applies here: the cost of enumerating all the aspects should be bound by a polynomial on the number of fragments (hence, on the number of granules).

(4) Validating all those efficiency constraints may be difficult, especially if appealing definitions have worst-case bounds that are not polynomial (or are polynomials of high degree, where for practical purposes anything above degree 2 or 3 is probably too high). One possibility (shown in this work) is to use standard statistical techniques on real programs, in the effort to figure out if the worst-case presents itself in practice.

(5) Now, I would organize the enumeration of aspects of fragments in such a way that one can eventually use in the parallelization of a fragment the results of having parallelized its subparts. This is the *eventual enumeration ordering*

property of the analysis. This property guarantees that, after inspecting a fragment for the last time, all of its super-fragments will still have at least one chance to be inspected (and, therefore, to exploit the previous results).

(6) Having organized this fixed search strategy (i.e., defining what granules, fragments and aspects will be, and providing iterators that satisfy the eventual enumeration ordering property), I would then design, separately, the rules of transformation to apply. To this end, I would consider the characterization of the fragments in terms of the data and control flow relationships, and express conditions to convert a piece of code into another with a larger execution set. The conditions are to be explicit, and assume nothing about the target machine. This is generally possible, as their correctness depends only on the semantics of the underlying language. The collection of these conditions and transformations is the *catalog*.

Experience with SPOIL has shown that one can conveniently put in the catalog transformations that either (1) don't increase any execution sets, or (2) exploit knowledge of target architecture. My first reaction would be to banish the first to the front end (and then define a condition-and-transformation language that let me query the front end when needed), and the second to the back end, where they properly belong. However, although my model sheds light on which transformations are properly called parallelizations, it is to be expected that practical implementations will compromise here and tolerate some blurring of this distinction, on account of efficiency.

The same is true of the standard compiler model, and those compromises (for instance, the merging of lexical analysis and parsing in one phase) are known not to hinder the use of the standard model to understand or design such compilers. It seems to me that the model presented in this dissertation, because of the flexibility involved in the various entities that need to be defined in the course of its use, will show the same robustness in the presence of compromises dictated by a pragmatic sense.

On the other hand, that very same flexibility makes it difficult to present testable claims about the adequacy of my model for actual parallelization work. After all, most program transformation schemes (even sequential compilation!) could be cast in terms close to the language of granules, fragments, aspects, etc. For this reason I presented specific instantiations of these concepts. SPOIL represents programs as graphs of either C statements, or of basic blocks. In addition, I provided a specific definition of fragment (in terms of depth-first-numbering intervals) and a specific definition of aspect (in terms of quotients of fragments with respect to their maximal subfragments). Then I provided an instantiation of the catalog notion, that uses only specific primitives of limited power (thereby not equivalent to arbitrary code). The combination of these instantiations is seen to be efficiently practical (at least in the statistical average case), and also capable to express the parallelizations in the literature. Indeed, the version of SPOIL that uses C-level statements is seen to be able to discover parallelism at both fine and coarse grains simultaneously, precisely because it is not unduly concerned with fitting a given architecture.

This evidence reinforces my claim that the general model exhibits, in its domain of competence, similar descriptive and prescriptive powers as the standard model does in its own.

7.2. Future Work

Many questions have remained open, or have been newly opened, in the course of this investigation. This section collects some of them.

First of all, a definitively persuasive demonstration of the usability of my model has to go beyond detection of parallelism, and enter into the exploitation domain. As in the sequential case, I would expect that the synthesis of runnable code be less amenable to a direct, theoretically principled attack, than the analysis has shown itself to be. Still, I would like to investigate ways to map information in the execution set hierarchy (plus the rest of the program representation) to runnable code.

There is also work to do in the front end. SPOIL does a very light job of data flow analysis, and the current implementation still uses queries to the user to answer some of the traditional data flow questions. Replacing the user with a real data flow analyzer would permit a better assessment of the effectiveness with which SPOIL exploits the available information. Also, I would like to determine how far can a simple catalog go when the data flow analyzer cannot obtain precise estimations because of aliasing.

Another point of interest is the possibility of using one of the contemporary combined representations of control and data flow relationships [CFR88, FOW87] instead of traditional control flow graphs augmented with data flow information, thereby further unifying the analysis and reducing the number of primitives needed in the catalog.

Another area of investigation is the exploration of other definitions of granule, fragment and aspect, with views to characterize the instantiations of these concepts that are effective in practice.

There is also the issue that SPOIL's current catalog is flat: each rule is tried in order for every aspect. Many subtests are repeated for the same aspect. A structured catalog would cut down substantially on the cost of the analysis. It also opens up the question (well known in program optimization) of determining optimal rule orderings. Perhaps some non-deterministic presentation of the guards will be needed.

Finally, the work discussed here belongs in the long tradition of program representation and analysis. I am interested in knowing whether it is possible to go from a formal representation of the semantics of a language to an automatic choice of definitions for granule, fragment, etc...

Bibliography

[ASU86] A. V. AHO, R. SETHI and J. D. ULLMAN, *Compilers: Principles, Techniques and Tools*, Addison-Wesley Publishing Company, 1986.

[Aik88] A. AIKEN, "Compaction-Based Parallelization", Ph.D. Thesis, Tech. Rep. 88–922, Department of Computer Science, Cornell University, Ithaca, NY 14853–7501, June 1988.

[AlC72] F. E. ALLEN and J. COCKE, "A Catalogue of Optimizing Transformations", in *Design and Optimization of Compilers*, R. RUSTIN (editor), Prentice Hall, Englewood Cliffs, NJ, 1972, 1–30. Courant Computer Science Symposium 5, May 1971.

[AlC76] F. ALLEN and J. COCKE, "A Program Data Flow Analysis Procedure", *Comm. of the ACM 19*, 3 (March 1976), 137–147, ACM.

[AlK82] J. R. ALLEN and K. KENNEDY, "PFC: A Program to Convert Fortran to Parallel Form", in *Proceedings IBM Conf. Parallel Computers in Scientific Computations*, Rome, 1982.

[AKP83] J. R. ALLEN, K. KENNEDY, C. PORTERFIELD and J. WARREN, "Conversion of Control Dependence to Data Dependence", *Conference Record of the Tenth ACM Symp. on Prin. of Prog. Lang.*, Austin, Texas, January 1983, 177–189.

[ABC87] F. ALLEN, M. BURKE, P. CHARLES, R. CYTRON and J. FERRANTE, "An Overview of the PTRAN Analysis System for Multiprocessing", 13115 (#56866), Computer Science Department, IBM T. J. Watson Research Center, Yorktown Heights, NY 10598, 29 September 1987.

[ACK87] R. ALLEN, D. CALLAHAN and K. KENNEDY, "Automatic Decomposition of Scientific Programs for Parallel Execution", *Conference Record of the Fourteenth ACM Symp. on Prin. of Prog. Lang.*, Munich, West Germany, January 1987, 63–76.

[AlJ88] R. ALLEN and S. JOHNSON, "Compiling C for Vectorization, Parallelization, and Inline Expansion", *SIGPLAN Notices: Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation 23*, 7 (July 1988), 241–249, ACM.

[AmS86] A. AMIR and C. H. SMITH, "The Syntax of Parallelism", UMIACS-Tech. Rep.-86-24, Institute for Advanced Computer Studies, University of Maryland, November 1986. Also CS-Tech. Rep.-1744, Department of Computer Science.

[AAG87] M. ANNARATONE, E. ARNOULD, T. GROSS, H. T. KUNG, M. LAM, O. MENZILCIOGLU and J. A. WEBB, "The Warp Computer: Architecture, Implementation and Performance", *Transactions on Computers C–36*, 2 (December 1987), IEEE.

[Ban79] U. BANERJEE, "Speedup of Ordinary Programs", UIUCDS-R-79-989, U of Illinois at Urbana-Champaign, October 1979.

[Ber66] A. J. BERNSTEIN, "Analysis of Programs for Parallel Processing", *IEEE Transactions on Electronic Computers EC-15*, 5 (1966), 757–763, IEEE.

[Bur87] M. BURKE, "An Interval-Based Approach to Exhaustive and Incremental Interprocedural Data Flow Analysis", 12702 (#56865), Computer Science Department, IBM T. J. Watson Research Center, Yorktown Heights, NY 10598, 1 September 1987.

[Cal88] D. CALLAHAN, "The Program Summary Graph and Flow-sensitive Interprocedural Data Flow Analysis", *SIGPLAN Notices: Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation 23*, 7 (July 1988), 47–56, ACM.

[CaH74] R. H. CAMPBELL and N. HABERMANN, "The specification of process synchronization by Path Expressions", in *LNCS*, vol. 16 , Springer-Verlag, 1974.

[Cam76] R. H. CAMPBELL, *Path Expressions: A technique for specifying process synchronization*, Ph.D. Dissertation, Comput. Lab., University of Newcastle-Upon-Tyne, August 1976.

[CoK88] K. D. COOPER and K. KENNEDY, "Interprocedural Side-Effect Analysis in Linear Time", *SIGPLAN Notices: Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation 23*, 7 (July 1988), 57–66, ACM.

[CoT90] W. R. COWELL and C. P. THOMPSON, "Tools to Aid in Discovering Parallelism and Localizing Arithmetic in Fortran Programs", *Software—Practice and Experience 20*, 1 (January 1990), 25–47, John Wiley & Sons.

[Cro91] L. A. CROWL, "Architectural Adaptability in Parallel Programming", Tech. Rep. 381, University of Rochester, Department of Computer Science, Ph.D. Thesis, May 1991.

[CyF88] R. CYTRON and J. FERRANTE, "An Improved Control-Dependence Algorithm", RC 13291 (#60163), IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10598, January 1988.

[CFR88] R. CYTRON, J. FERRANTE, B. K. ROSEN, M. N. WEGMAN and F. K. ZADECK, "An Efficient Method of Computing Static Single Assignment Form", CS-88-16, Brown University, Department of Computer Science, Providence, Rhode Island 02912, October 1988. Also in 16th ACM Symp. on Prin. of Prog. Lang., 1989.

[DaF84] J. W. DAVIDSON and C. FRASER, "Register Allocation and Exhaustive Peephole Optimization", *Software—Practice & Experience 14*, 9 (September 1984), 857–866, John Wiley & Sons.

[Ell86] J. R. ELLIS, *Bulldog: A compiler for VLIW Architectures*, MIT Press, 1986. Also Tech. Rep. YALEU/DCS-RR-364. Originally a Yale University Ph.D. Thesis, February 1985.

[FKZ76] R. FARROW, K. KENNEDY and L. ZUCCONI, "Graph Grammars and Global Program Data Flow Analysis", *Proc. 17th Annual Symp. on Foundations of Computer Science*, Houston, 1976, 42–56.

[FeO83] J. FERRANTE and K. J. OTTENSTEIN, "A Program Form Based on Data Dependency in Predicate Regions", *Conference Record of the Tenth ACM Symp. on Prin. of Prog. Lang.*, Austin, Texas, January 1983, 217–236.

[FOW87] J. FERRANTE, K. J. OTTENSTEIN and J. D. WARREN, "The Program Dependence Graph and Its Use in Optimization", *Trans. Prog. Lang and Systems 9*, 3 (July 1987), 319–349, ACM.

[Fis83] J. A. FISHER, "Very Long Instruction Word Architectures and the ELI-512", *Tenth Annual Symposium on Computer Architecture*, June 1983, 140–150.

[FoR72] C. C. FOSTER and E. M. RISEMAN, "The Inhibition of Potential Parallelism by Conditional Jumps", *IEEE Transactions on Computers 21*, 12 (1972), 1405–1411, IEEE.

[Gil58] S. GILL, "Parallel Programming", *The Computer Journal 1* (April 1958), 2–10, The British Computing Society.

[Gol87] R. GOLDBLATT, *Logics of Time and Computation*, Center for the Study of Language and Information, 1987.

[Har86] W. L. HARRISON, "Compiling Lisp for Evaluation on a Tightly Coupled Multiprocessor", CSRD Rpt. No. 565, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, Urbana, IL 61801-2932, 20 March 1986.

[HeU72] M. S. HECHT and J. D. ULLMAN, "Flow Graph Reducibility", *SIAM J. Comput. 1*, 2 (June 1972), 188–202.

[HeU75] M. S. HECHT and J. D. ULLMAN, "A Simple Algorithm for Global Data Flow Analysis", *SIAM Journal on Computing 4*, 4 (December 1975), 519–532, SIAM.

[HRB88] S. HORWITZ, T. REPS and D. BINKLEY, "Interprocedural Slicing Using Dependence Graphs", *SIGPLAN Notices: Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation 23*, 7 (July 1988), 35–46, ACM.

[Jed87] J. JEDRZEJOWICZ, "Nesting of Shuffle Closure Is Important", *Information Processing Letters 25* (26 July 1987), 363–367, North-

Holland Publishing Company.

[Jed88] J. JEDRZEJOWICZ, ''Infinite Hierarchy of Expressions Containing Shuffle Closure Operator'', *Information Processing Letters 28* (30 May 1988), 33–37, North-Holland Publishing Company.

[Kas75] V. N. KAS'JANOV, ''Distinguishing Hammocks in a Directed Graph'', *Soviet Math. Doklady 16*, 5 (1975), 448–450, American Mathematical Society.

[Kas73] V. N. KASYANOV, ''Some Properties of Fully Reducible Graphs'', *Information Processing Letters 2* (1973), 113–117, North-Holland Publishing Company.

[Ken80] K. KENNEDY, ''Automatic translation of Fortran programs to vector form'', 476 029 4, Rice U., October 1980.

[KeR78] B. W. KERNIGHAN and D. M. RITCHIE, *The C Programming Language*, Prentice-Hall, 1978.

[KeR89] B. W. KERNIGHAN and D. M. RITCHIE, *The C Programming Language*, Prentice-Hall, 1989. Second Edition.

[Kuc75] D. J. KUCK, ''Parallel Processing of Ordinary Programs'', UIUCDCS-R-75-767, University of Illinois at Urbana-Champaign Dept. of C. Sc., November 1975.

[KuM84] H. T. KUNG and O. MENZILCIOGLU, *Design Specifications for the CMU Warp Processor*, Carnegie-Mellon University, 1984.

[Lam88] M. LAM, ''Software Pipelining: An Effective Scheduling Technique for VLIW Machines'', *SIGPLAN Notices: Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation 23*, 7 (July 1988), 318–328, ACM.

[Lam89] M. S. LAM, *A Systolic Array Optimizing Compiler*, Kluwer Academic Publishers, 1989.

[Lam86] L. LAMPORT, ''The Mutual Exclusion Problem Part I — A Theory of Interprocess Communication'', *J. ACM 33*, 2 (April 1986), 331–326, ACM.

[Lar89] J. R. LARUS, *Restructuring Symbolic Programs for Concurrent Execution on Multiprocesso* Computer Science Division (EECS), University of Berkeley, Berkeley, CA 94720, May 1989. Ph.D. Thesis; also report UCB/Computer Science Dpt. 89/502.

[LBR81] M. LEPAGE, D. BARNARD and A. RUDMIK, ''Optimization of Hierarchical Directed Graphs'', *Computer Languages 6* (1981), 19–34, Pergammon Press Ltd.

[Li88] Z. LI and P. YEW, ''Efficient Interprocedural Analysis for Program Parallelization and Restructuring'', *SIGPLAN Notices 23*, 9 (September 1988), 85–99, ACM Press. Proceedings of the ACM/SIGPLAN PPEALS 1988, 19–21 July 1988.

[Mil85] G. J. MILNE, ''CIRCAL and the Representation of Communication, Concurrency and Time'', *ACM Transactions on Programming Languages and Systems 7*, 2 (April 1985), 270–298, ACM.

[Ney88] A. D. NEYRINCK, ''Static Analysis of Aliases and Side Effects in Higher-Order Languages'', 88–896, Cornell University, Department of Computer Science, Ithaca, NY 14853-7501, February 1988. Ph.D. Thesis. January 1988 is also given as a date.

[Nic85] A. NICOLAU, ''Uniform Parallelism Extraction in Ordinary Programs'', *Proceedings of the ICPP*, 1985, 614–618.

[PHK88] D. PADUA, W. L. HARRISON and D. KUCK, ''Machines, Languages and Compilers for Parallel Symbolic Computing'' CSRD 729, Center for Supercomputing Research and Development, University of Illinois, September 1988. (Also in *Biological and Artificial Intelligence Systems*, E. Clementy and S. Chin, Eds. ESCOM, Leiden, The Netherlands, 1988. Pp. 453–461.).

[Pol88] C. D. POLYCHRONOPOULOS, *Parallel Programming and Compilers*, Kluwer Academic Publishers, 1988.

[RaG69] C. V. RAMAMOORTHY and M. J. GONZALEZ, ''A survey of techniques for recognizing parallel processable streams in computer programs'', *1969 Fall Joint Computer Conf. 35* (1969), 1–15, AFIPS press.

[Sha80] M. SHARIR, ''Structural Analysis: A New Approach To Flow Analysis in Optimizing Compilers'', *Computer Languages 5* (1980), 141–153, Pergammon Press Ltd.

[Sha78] A. C. SHAW, ''Software Descriptions with Flow Expressions'', *Transactions in Software Engineering E–4*, 3 (1978), 242–254, IEEE.

[Shi88] O. SHIVERS, ''Control Flow Analysis in Scheme'', *SIGPLAN Notices: Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation 23*, 7 (July 1988), 164–174, ACM.

[SmA88] K. SMITH and W. F. APPELBE, ''PAT -- An Interactive Fortran Parallelizing Assistant Tool'', *Proceedings of 1988 ICPP 2* (15–19 August 1988), 58–62, The Pennsylvania State University Press.

[Sta89] R. M. STALLMAN, *Using and Porting GCC*, Free Software Foundation. 675 Mass Ave, Cambridge, MA 02139, 1989.

[Ste84] G. L. STEELE, *Common Lisp*, Digital Press, 1984. Second, much revised edition, 1990.

[Tar74] R. E. TARJAN, ''Testing Flow Graph Reducibility'', *Journal of Computer and System Sciences 9* (1974), 355–365.

[TjF70] G. S. TJADEN and M. J. FLYNN, ''Detection and parallel execution of independent instructions'', *IEEE Transactions on Computers 21*, [illegible] (October 1970), 889–895.

[Vee85] A. H. VEEN, *The Misconstrued Semicolon*, Technische Hogeschool Eindhoven, Ph.D. Thesis, September 1985.

[Wad80] A. B. WADIA, ''Generation of Node Lists Using Segment Analysis'', *Computer Languages 5* (1980), 115–129, Pergammon Press Ltd.

[WaG83] W. M. WAITE and G. GOOS, *Compiler Construction*, Springer-Verlag, 1983.

[War84] J. WARREN, ''A Hierarchical Basis for Reordering Transformations'', *Conference Record of the Eleventh ACM Symp. on Prin. of Prog. Lang.*, Salt Lake City, Utah, January 1984, 272–282.

[Weg83] M. WEGMAN, ''Summarizing Graphs by Regular Expressions'', *Conference Record of the Tenth ACM Symp. on Prin. of Prog. Lang.*, Austin, Texas, January 1983, 203–216.

[Wol89] M. WOLFE, *Optimizing Supercompilers for Supercomputers*, The MIT Press, 1989. Based on Wolfe's Ph.D. dissertation, 1982.

[YuH85] T. YUASA and M. HAGIYA, *Kyoto Common Lisp Report*, Research Institute for Mathematical Sciences, Kyoto University, 1985.

APPENDIX A

A Catalog of Parallelizations

Here I collect the rules of parallelization discussed in the text. First comes an enumeration of the operations available to the transformer in SPOIL; then comes a catalog of rules.

The operations available to the transformer are expressed in S-expression notation (just because SPOIL is written in Lisp). The arguments of these forms are named either in the form **<name** or **>name**. In the first case, it is assumed that either a constant is used in the rule, or a variable appears in that position, and that variable is expected to be bound at the time this part of the guard is attempted. Failure to do so causes the test to fail. In the second case, the name has to be a variable that has to be unbound, or another S-expression containing unbound variables. This is the mechanism whereby tests and transformations record information for others in the same rule.

Elementary tests

These tests query for the correspondence of granules, fragments and aspects. Typically they are used early in a rule, to gain access to the material to be tested.

`(:aspect <fid <composition >variable)`

Given a fragment id *fid* and a list of fragment ids *composition*, these test binds the variable to a representation of the aspect of *fid* under quotient by *composition*.

`(:graph <aspect >variable)`

Given an aspect (as produced by the **`:aspect`** test), the variable is bound to a representation of the graph that remains after all the collapses.

`(:code <graph <variable)`

Given a graph, as produced by **`:graph`**, this test collects the instructions into a list (thereby mapping aspects to pieces of the original program).

`(:parse <graph <grammar <variable)`

Given a graph and a graph grammar, this test obtains the parse of the graph with respect to the grammar. In SPOIL there is only one grammar at all times, so the second subform is ignored. I have put strings in the examples, as comments. The **`<variable`** is usually a pattern that contains other

`(:part <parsevar <aspect >fragment-id)`

Given a variable that presumably was involved in a third subform of the **`:parse`** test and an aspect, this test succeeds if the subgraph of the fragment derived from the parse variable coincides with one of the fragments, in whose case the third subform is bound to its fragment id. This test appears in guards like 'if something is a loop, and has as its body a decision between two loops', where just being parsable is not all we care about.

(:execset <fid >variable)
Given a fragment id, this binds the variable to the S-expression that represents the execution set. Typically this will be a **:choice**.

(:exists-execset <fid <pattern)
This is more discriminating. It tries to match the current execution set of *fid* against the pattern, non-deterministically selecting a **:choice** that matches the pattern.

(:flow-attribute <fid <composition <attribute >variable)
Given a fragment and a composition list (a list of subfragments of it), this test queries the data flow information associated with the attribute. If known, or computable, the test succeeds and the variable is bound to the value. This test acts as the interface with front end in SPOIL, as almost the only thing SPOIL ever asks of the front end is data flow information.

(:flow-relation <fid0 <fid1 <fid2 <relation)
This is a more specific query. The test is whether the RELATION holds between FID1 and FID2 (two fragment ids) when seen in the context of FID0. This can only succeed or fail. Useful when checking whether specific dependence patterns occur.

(:match <pattern1 <pattern2)
A generic test that unifies both patterns. Often used to pick up pieces of a result from other tests.

(:intersection <set <set >variable)

(:union <set <set >variable)

(:setdiff <set <set >variable)

(:singleton <set)

(:subset <set <set)

(:emptyset <set)

(:sameset <set <set)
The data flow information returned by **:flow-attribute** is normally expressed as a set (of locations, of variables, ...). The first three succeed when the specified set operation among the given sets is possible (i.e., whenever the first two subforms are bound to constants), and then binds the third subform (must be a variable) to the result. The rest don't bind any variables, just succeed or fail according to whether the relation holds.

(:not <guard)

(:and <guard ...)

(:or <guard ...)
Guards are defined recursively. These three forms succeed when the logical relationship implied by their names holds. The resulting bindings are as

follows: All the bindings generated by all the branches of **`:and`** are available; all the bindings generated by one of the branches of **`:or`** are available (which branch is decided in SPOIL by choosing the first branch to succeed); and no branches at all from the subform of **`:not`** are available.

`(:map <guard <variable1 <fsetlist >variable2)`
FSETLIST must be bound to a list of fragments. VARIABLE1 is bound to each of them in turn. For each such binding, the GUARD is tested. If the GUARD succeeds, the value of VARIABLE1 is added to a list. At the end, all the values of VARIABLE1 that satisfied the recursive GUARD are in the list, which is then bound to the VARIABLE2.

`(:exists <guard <variable1 <fsetlist >variable2)`
This is similar to **`:map`**, but possibly useful more often. As soon as one binding of VARIABLE1 in FSETLIST satisfies GUARD, that binding is given to VARIABLE2 and the guard succeeds.

`(:forall <guard <variable <fsetlist)`
This is similar to **`:exists`**. This test fails if any binding of VARIABLE to the elements of FSETLIST does not satisfy the GUARD. Else it succeeds, but leaves no bindings to be observed.

`(:guard <guard <fragment <fid <composition <pgraph)`
This is the recursive form of the tests.

`(:oracle <guard)`
This is a back-door used when testing new queries that haven't been implemented yet. It just asks the user to provide the information that the transformer needs.

Transformations

`(:match <pattern1 <pattern2)`
As in the case of guards, this is just a generic unification pattern matcher.

`(:new-code <codelist <at-fid >fid);?`
CODELIST contains a list of instructions (RTL level), it is either a constant or was produced via the guard test **`:code`**. A new fragment is built on that code, the transformer binds the variable in the third subform to the new fragment id. The second subform is a pre-existing fragment id. The new fragment is considered a re-interpretation of the fragment at that position in the hierarchy (essentially, this means that both fragments share their execution sets). This is the mechanism used in SPOIL to perform reorganization and compensation.

`(:new-name >new-name)`
This transformation generates a new name (useful to rename variables or labels in new RTL code).

`(:with-new-name <oldname <newname <oldcodelist >newcodelist)`
Replace the NEWNAME (possibly generated by **`:new-name`**, but could also be a constant symbol) for all occurrences of OLDNAME in OLDCODELIST, yielding NEWCODELIST.

(:new-execset <eset >evariable)
Given an S-expression representing an execution set, bind the EVARIABLE to it.

(:combine-execsets <esettag <esetlist >eset)
Given several execution sets in a list (possibly created by **:map**), make a new one whose top level operator is ESETTAG.

(:add-to-execset <fid <new-choice)
Replace the execution set associated with FID with a choice between the old value and the NEW-CHOICE.

(:apply <transform ...)
Simply execute in order all the transforms given by the subforms.

(:map <transform <invar <inlist <outvar >outlist)
Bind INVAR to each fragment in INLIST, apply the TRANSFORM. If OUTVAR is bound by the application, collect the value in OUTLIST.

A Catalog

Independent Sequence

```
(catalog.add-rule
 cat
 "Independent Sequence"             ;using Bernstein's conditions
 (guard.new '(:and
              (:aspect ?f ?c ?a)
              (:graph ?a ?pg)
              (:parse ?pg "sequence" ?fparse)
              (:match ?fparse (:cc (:cc ?first ?tail) ?exit))
              (:part ?first ?a ?f.Seq.First)
              (:part ?tail  ?a ?f.Seq.Tail)
              (:flow-attribute ?f.Seq.First () :use ?Use.f.Seq.First)
              (:flow-attribute ?f.Seq.Tail  () :use ?Use.f.Seq.Tail)
              (:flow-attribute ?f.Seq.First () :def ?Def.f.Seq.First)
              (:flow-attribute ?f.Seq.Tail  () :def ?Def.f.Seq.Tail)
              (:intersection ?Def.f.Seq.First ?Use.f.Seq.Tail ?1)
              (:intersection ?Use.f.Seq.First ?Def.f.Seq.Tail ?2)
              (:intersection ?Def.f.Seq.First ?Def.f.Seq.Tail ?3)
              (:emptyset ?1)        ;not flow dependent
              (:emptyset ?2)        ;not anti dependent
              (:emptyset ?3)        ;not output dependent
              ))
 (transform.new '(:apply
                  (:add-to-execset ?f
                   (:parallel
                    (:execset ?f.Seq.First)
                    (:execset ?f.Seq.Tail))))))
```

Forced Series

```
(catalog.add-rule
 cat
 "Forced Series"
 (guard.new '(:and
             (:aspect ?f ?c ?a)
             (:graph ?a ?pg)
             (:parse ?pg "sequence" ?fparse)
             (:match ?fparse (:cc (:cc ?first ?tail) ?exit))
             (:part ?first ?a ?f.Seq.First)
             (:part ?tail  ?a ?f.Seq.Tail)
             (:flow-attribute ?f.Seq.First () :use ?Use.f.Seq.First)
             (:flow-attribute ?f.Seq.Tail  () :use ?Use.f.Seq.Tail)
             (:flow-attribute ?f.Seq.First () :def ?Def.f.Seq.First)
             (:flow-attribute ?f.Seq.Tail  () :def ?Def.f.Seq.Tail)
             (:intersection ?Def.f.Seq.First ?Use.f.Seq.Tail ?1)
             (:intersection ?Use.f.Seq.First ?Def.f.Seq.Tail ?2)
             (:intersection ?Def.f.Seq.First ?Def.f.Seq.Tail ?3)
             (:or
              (:not (:emptyset ?1)) ;flow dependent
              (:not (:emptyset ?2)) ;anti dependent
              (:not (:emptyset ?3)) ;output dependent
             )))
 (transform.new '(:apply
                 (:add-to-execset ?f
                  (:series
                   (:execset ?f.Seq.First)
                   (:execset ?f.Seq.Tail))))))
```

Loop Recognition

```
(catalog.add-rule
 cat
 "Loop Recognition"
 (guard.new '(:and
             (:aspect ?f ?c ?a)
             (:match ?c (?LoopBody))
             (:graph ?a ?fpg)
             (:parse ?fpg "loop ?fparse)
             (:match ?fparse (:cc (:loop ?Body) ?Exit))))
 (transform.new '(:apply
                 (:add-to-execset ?f
                  (:sloop (:execset ?LoopBody))))))
```

Loop Distribution

```
(catalog.add-rule
 cat
 "Loop Distribution"
 (guard.new '(:and
              (:execset ?f ?e)
              (:match ?e (:sloop (:parallel . ?parts)))))
 (transform.new '(:apply
                  (:map (:new-execset (:sloop ?part) ?newloop)
                   ?part ?parts
                   ?newloop ?newloops)
                  (:combine-execsets :parallel ?newloops ?neweset)
                  (:add-to-execset ?f ?neweset))))
```

DO to DOALL

```
(catalog.add-rule                          ;1-loops/swap
 cat
 "SLoop -> PLoop (DO -> DOALL)"
 (guard.new '(:and
              (:exists-execset ?f (:sloop ?body))
              (:match ?body (:execset ?fb))
              (:flow-relation ?f ?fb ?fb :loop-independent)))
 (transform.new '(:apply
                  (:new-execset (:ploop ?body) ?newloop)
                  (:add-to-execset ?f ?newloop))))
```

Partial Distribution

```
(catalog.add-rule                          ;polyc/loop3
 cat
 "(SLOOP X . REST) -> (SERIES (PLOOP X) (SLOOP REST))"
 (guard.new '(:and
              (:exists-execset ?f (:sloop ?x . ?rest))
              (:match ?x (:execset ?fx))
              (:flow-relation ?f ?fx ?fx :loop-independent)))
 (transform.new '(:apply
                  (:new-execset (:series
                                  (:ploop ?x)
                                  (:sloop . ?rest)) ?newloop)
                  (:add-to-execset ?f ?newloop))))

(catalog.add-rule                          ;polyc/loop2
 cat
 "(SLOOP REST X) -> (SERIES (SLOOP REST) (PLOOP X))"
 (guard.new '(:and
              (:exists-execset ?f (:sloop ?rest ?x))
              (:match ?x (:execset ?fx))
              (:flow-relation ?f ?fx ?fx :loop-independent)))
 (transform.new '(:apply
                  (:new-execset (:series
                                 (:sloop ?rest)
                                 (:ploop ?x)) ?newloop)
                         (:add-to-execset ?f ?newloop))))
```

DOACROSS

```
(catalog.add-rule                         ;wolfe/doacr
 cat
 "DOACROSS"
 (guard.new '(:and
               (:exists-execset ?f (:sloop . ?loopbody))
               (:match ?c (?body))    ;only when the entire body is visible
               (:exists-execset ?body (:series . ?loopbody))
               (:flow-attribute ?f (?body) :synchronized-code ?synch-code)))
 (transform.new '(:apply
                   (:map (:apply
                           (:match ?synch (?fid ?newcode))
                           (:new-code ?newcode ?fid ?nfid)
                           (:new-execset (:execset ?nfid) ?enfid))
                    ?synch ?synch-code ?enfid ?enfids)
                   (:combine-execsets :ploop ?enfids ?new-eset)
                   (:add-to-execset ?f ?new-eset))))
```

Ignorable Self Anti-dependences

```
(catalog.add-rule                              ;wolfe p.64
 cat
 "Ignorable self-antidependence cycles"
 (guard.new '(:and
               (:exists-execset ?f (:sloop (:execset ?fb)))
               (:flow-relation ?f ?fb ?fb :flow-dependent)
               (:not (:flow-relation ?f ?fb ?fb :anti-dependent))
               (:not (:flow-relation ?f ?fb ?fb :output-dependent))
               (:flow-attribute ?fb () ?def-locs ?d)
               (:match ?d (?target))
               (:code ?fb ?oldcode)))
 (transform.new '(:apply
                   (:new-name ?new-array)
                   (:new-code ((call-insn 'copy ?target ?new-array))
                    nil ?copy-in)
                   (:with-new-name ?target ?new-array ?oldcode ?newcode)
                   (:new-code ?newcode ?newbody)
                   (:new-code ((call-insn 'copy ?new-array ?target))
                    nil ?copy-out)
                   (:new-execset (:series
                                   ?copy-in
                                   (:ploop ?newbody)
                                   ?copy-out)
 ?new-eset)
(:add-to-execset ?f ?new-eset))))
```

APPENDIX B

Example Programs

Test Suite

A number of programs were used during the testing phase of SPOIL. They are collected here for ease of reference.

minmax

This program was originally the synthetic example I used to provide details on my work. It contains mostly mid-to-coarse level parallelism, in the form of independent reductions of minimum and maximum. The same loci of parallelization are open, of course, to "telescopic" reductions, where only a logarithmic number of iterations is needed. In its simplicity, it encodes most of the typical characteristics of promising parallelizable code.

```
/*
 - minmax(int x[], int n, int *min, int *max)
 *
 * X has indices 0..N. At the end, the minimum and the maximum in X are
 * returned through the MIN and MAX pointers.
 *
 * $Id: minmax.c,v 1.1 89/09/25 10:28:38 quiroz Exp Locker: quiroz $
 */

void
minmax(int x[], int n, int *min, int *max)
{
    int         i;                          /* loop index */
    int         localmin, localmax;         /* temporaries */

    localmin = localmax = x[0];

    i = 1;
    while ( i <= n ) {
      if (x[i] < localmin)
          localmin = x[i];
      if (x[i] > localmax)
          localmax = x[i];
      i += 1;
    }
```

```
        *min = localmin;
        *max = localmax;
        return;
}
```

minmaxsum

This is just the **minmax** program, with a little extra activity to calculate also the sum. Its reason for being was to provide a data point of a different size when the cost of fragment and aspect construction was still an issue.

```
/*
 - minmaxsum(int x[], int n, int *min, int *max, int *sum)
 *
 * X has indices 0..N. At the end, the minimum, the maximum, and the
 * sum of X are returned through the MIN, MAX, and SUM pointers.
 *
 * $Id: minmax.c,v 1.1 89/09/25 10:28:38 quiroz Exp Locker: quiroz $
 */

void
minmaxsum(int x[], int n, int *min, int *max, int *sum)
{
        int                 i;                      /* loop index */
        int                 localmin, localmax; /* temporaries */
        int             localsum;

        localmin = localmax = x[0];
        localsum = 0;

        i = 1;
        while ( i <= n ) {
          if (x[i] < localmin)
              localmin = x[i];
          if (x[i] > localmax)
              localmax = x[i];
            if (x[i] > 0)
                localsum += x[i];               /* hmmm... */
          i += 1;
        }

        *min = localmin;
        *max = localmax;
        *sum = localsum;
        return;
}
```

loop1

The following three programs come, essentially, from [Pol88, sec. 2.3.1, p. 22].

```
/* 2.3.1, p.22 up */

/* S1 and S2 can be vectorized independently */

void
oneloop(a, b, c, d, n, k)
    double          *a, *b, *c, *d;
    int             n;
    double          k;
{
    int             i;

    for (i = 0; i < n; i++) {
        a[i] = b[i] + c[i];             /* S1 */
        d[i] = b[i] * k;                /* S2 */
    }
}
```

loop2

```
/* 2.3.1, p.22 down */

/* S3 can be taken out of the loop and vectorized :

   (sloop H . Rest) -> (series (ploop H) (sloop Rest))
   (sloop Rest | (T)) -> (series (sloop Rest) (sloop T))

   where H (or T) don't carry any dependencies.
*/

void
loop(a, b, c, n, k)
    double          *a, *b, *c;
    int             n;
    double          k;
{
    int             i;

    for (i = 0; i < n; i++) {
        a[i+1] = b[i-1] + c[i];         /* S1 */
        b[i]   = a[i] * k;              /* S2 */
        c[i]   = b[i] - 1;              /* S3 */
    }
}
```

loop3

```
/* 2.3.1, p.22 down--MODIFIED */

/* S3 can be taken out of the loop and vectorized :

   (sloop H . Rest) -> (series (ploop H) (sloop Rest))
   (sloop Rest | (T)) -> (series (sloop Rest) (sloop T))

   where H (or T) don't carry any dependencies.
*/

void
loop(a, b, c, n, k)
    double          *a, *b, *c;
    int             n;
    double          k;
{
    int             i;

    for (i = 0; i < n; i++) {
        c[i]   = b[i] - 1;              /* S3 */
        a[i+1] = b[i-1] + c[i];         /* S1 */
        b[i]   = a[i] * k;              /* S2 */
    }
}
```

wolfe-4.2

This program is a translation of the example given in [Wol89, sec. 4.2, p. 74].

```
/*
 * Adapted from Wolfe's Optimizing Supercompilers for Supercomputers, p. 74
 */

#define N 100

void
example(int a[N][N], int b[N][N])
{
    int             i, j;

    /* Summary:
     *
     * S1 data(<,<) S2
     * S1 anti(=,=) S2
     *
     * So the inner loop doesn't carry the antidependence, and the inner loo
     * doesn't see it when executed serially.
     *
     */


    for (i = 2; i < N; i++)                /* serial loop */
        for (j = 2; j < N; j++) {          /* parallel loop */
/*S1*/      a[i][j] = b[i][j] + 2;
/*S2*/      b[i][j] = a[i-1][j-1] - b[i][j];
        }
}
```

doacr

This program comes from [Wol89, sec. 4.3, p. 76].

```
/* Wolfe, p. 76 */

void
doacr(a, b, c, d, e, n)
    int             n;
    double          *a, *b, *c, *d, *e;
{
    int             i;

    for (i = 1; i < n; i++) {
        /* wait(S3, I-1) */
        a[i] = b[i] + c[i-1];          /* S1 */
        /* post(S1, I) */
        d[i] = a[i] * 2.0;             /* S2 */
        /* wait(S1, I-1) */
        c[i] = a[i-1] + c[i];          /* S3 */
        /* post(S3, I) */
        /* wait(S3, I-2) */
        e[i] = d[i] + c[i-2];          /* S4 */
    }
}
```

decomp

The next two programs form a linear system solver, taken from the NETLIB archive. The original (FORTRAN) versions are from George Forsythe, Mike Malcolm, and Cleve Moler. Those FORTRAN versions were translated by the Fortran to C service also available at NETLIB. For more information on NETLIB, send electronic mail to `netlib@research.att.com`. A message body that reads simply:

```
    send index
```

will provide the necessary information.

```
/*  -- translated by f2c (version of 27 April 1990  12:27:33).
   You must link the resulting object file with the libraries:
      -lF77 -lI77 -lm -lc   (in that order)
*/

#include "f2c.h"

/* Subroutine */ int decomp_(ndim, n, a, cond, ipvt, work)
integer *ndim, *n;
doublereal *a, *cond;
integer *ipvt;
doublereal *work;
{
    /* System generated locals */
    integer a_dim1, a_offset, i_1, i_2, i_3;
    doublereal d_1, d_2;

    /* Local variables */
    static integer i, j, k, m;
    static doublereal t, anorm;
    extern /* Subroutine */ int solve_();
    static doublereal ynorm, znorm;
    static integer kb;
    static doublereal ek;
    static integer km1, nm1, kp1;



/*     decomposes a double precision matrix by gaussian elimination */
/*     and estimates the condition of the matrix. */

/*     use solve to compute solutions to linear systems. */

/*     input.. */

/*        ndim = declared row dimension of the array containing  a. */

/*        n = order of the matrix. */

/*        a = matrix to be triangularized. */

/*     output.. */

/*        a  contains an upper triangular matrix  u  and a permuted */
/*          version of a lower triangular matrix  i-l  so that */
/*          (permutation matrix)*a = l*u . */
/*        cond = an estimate of the condition of  a . */
/*           for the linear system  a*x = b, changes in  a  and  b */
/*           may cause changes  cond  times as large in  x . */
/*           if  cond+1.0 .eq. cond , a is singular to working */
/*           precision.  cond  is set to  1.0d+32  if exact */
/*           singularity is detected. */

/*        ipvt = the pivot vector. */
/*           ipvt(k) = the index of the k-th pivot row */
/*           ipvt(n) = (-1)**(number of interchanges) */

/*     work space..  the vector  work  must be declared and included */
/*                   in the call.  its input contents are ignored. */
/*                   its output contents are usually unimportant. */

/*     the determinant of a can be obtained on output by */
/*        det(a) = ipvt(n) * a(1,1) * a(2,2) * ... * a(n,n). */


    /* Parameter adjustments */
    --work;
    --ipvt;
    a_dim1 = *ndim;
    a_offset = a_dim1 + 1;
    a -= a_offset;

    /* Function Body */
    ipvt[*n] = 1;
    if (*n == 1) {
      goto L80;
    }
    nm1 = *n - 1;

/*     compute 1-norm of a */

    anorm = 0.;
    i_1 = *n;
    for (j = 1; j <= i_1; ++j) {
      t = 0.;
      i_2 = *n;
      for (i = 1; i <= i_2; ++i) {
          t += (d_1 = a[i + j * a_dim1], abs(d_1));
/* L5: */
      }
      if (t > anorm) {
          anorm = t;
```

```
 }
/* L10: */
    }

/*     gaussian elimination with partial pivoting */

    i_1 = nm1;
    for (k = 1; k <= i_1; ++k) {
      kp1 = k + 1;

/*         find pivot */

      m = k;
      i_2 = *n;
      for (i = kp1; i <= i_2; ++i) {
          if ((d_1 = a[i + k * a_dim1], abs(d_1)) > (d_2 = a[m + k * a_dim1]
                , abs(d_2))) {
            m = i;
          }
/* L15: */
      }
      ipvt[k] = m;
      if (m != k) {
          ipvt[*n] = -ipvt[*n];
      }
      t = a[m + k * a_dim1];
      a[m + k * a_dim1] = a[k + k * a_dim1];
      a[k + k * a_dim1] = t;

/*         skip step if pivot is zero */

      if (t == 0.) {
          goto L35;
      }

/*         compute multipliers */

      i_2 = *n;
      for (i = kp1; i <= i_2; ++i) {
          a[i + k * a_dim1] = -a[i + k * a_dim1] / t;
/* L20: */
      }

/*         interchange and eliminate by columns */

      i_2 = *n;
 for (j = kp1; j <= i_2; ++j) {
          t = a[m + j * a_dim1];
          a[m + j * a_dim1] = a[k + j * a_dim1];
          a[k + j * a_dim1] = t;
          if (t == 0.) {
            goto L30;
          }
          i_3 = *n;
          for (i = kp1; i <= i_3; ++i) {
            a[i + j * a_dim1] += a[i + k * a_dim1] * t;
/* L25: */
          }
L30:
          ;
     }
L35:
     ;
   }

/*     cond = (1-norm of a)*(an estimate of 1-norm of a-inverse) */
/*     estimate obtained by one step of inverse iteration for the */
/*     small singular vector.  this involves solving two systems */
/*     of equations, (a-transpose)*y = e  and  a*z = y  where  e */
/*     is a vector of +1 or -1 chosen to cause growth in y. */
/*     estimate = (1-norm of z)/(1-norm of y) */

/*     solve (a-transpose)*y = e */

    i_1 = *n;
    for (k = 1; k <= i_1; ++k) {
      t = 0.;
      if (k == 1) {
          goto L45;
      }
      km1 = k - 1;
      i_2 = km1;
      for (i = 1; i <= i_2; ++i) {
          t += a[i + k * a_dim1] * work[i];
/* L40: */
      }
L45:
      ek = 1.;
      if (t < 0.) {
          ek = -1.;
      }
      if (a[k + k * a_dim1] == 0.) {
```

```
	goto L90;
	 }
	 work[k] = -(ek + t) / a[k + k * a_dim1];
/* L50: */
     }
     i_1 = nm1;
     for (kb = 1; kb <= i_1; ++kb) {
       k = *n - kb;
       t = 0.;
       kp1 = k + 1;
       i_2 = *n;
       for (i = kp1; i <= i_2; ++i) {
           t += a[i + k * a_dim1] * work[i];
/* L55: */
       }
       work[k] = t + work[k];
       m = ipvt[k];
       if (m == k) {
           goto L60;
       }
       t = work[m];
       work[m] = work[k];
       work[k] = t;
L60:
       ;
     }

    ynorm = 0.;
    i_1 = *n;
    for (i = 1; i <= i_1; ++i) {
      ynorm += (d_1 = work[i], abs(d_1));
/* L65: */
    }

/*     solve a*z = y */

    solve_(ndim, n, &a[a_offset], &work[1], &ipvt[1]);

    znorm = 0.;
    i_1 = *n;
    for (i = 1; i <= i_1; ++i) {
      znorm += (d_1 = work[i], abs(d_1));
/* L70: */
    }

/*     estimate condition */
```

```
    *cond = anorm * znorm / ynorm;
    if (*cond < 1.) {
      *cond = 1.;
    }
    return 0;

/*     1-by-1 */

L80:
    *cond = 1.;
    if (a[a_dim1 + 1] != 0.) {
      return 0;
    }

/*     exact singularity */

L90:
    *cond = 1e32;
    return 0;
} /* decomp_ */

#ifdef uNdEfInEd
comments from the converter:  (stderr from f2c)
   decomp:
Warning on line 168: local variable dsign never used
#endif
```

solve

```
/*  -- translated by f2c (version of 27 April 1990  12:27:33).
   You must link the resulting object file with the libraries:
      -lF77 -lI77 -lm -lc   (in that order)
*/

#include "f2c.h"

/* Subroutine */ int solve_(ndim, n, a, b, ipvt)
integer *ndim, *n;
doublereal *a, *b;
integer *ipvt;
{
    /* System generated locals */
    integer a_dim1, a_offset, i_1, i_2;

    /* Local variables */
    static integer i, k, m;
    static doublereal t;
    static integer kb, km1, nm1, kp1;


/*   solution of linear system, a*x = b . */
/*   do not use if decomp has detected singularity. */

/*   input.. */

/*     ndim = declared row dimension of array containing a . */

/*     n = order of matrix. */

/*     a = triangularized matrix obtained from decomp . */

/*     b = right hand side vector. */

/*     ipvt = pivot vector obtained from decomp . */

/*   output.. */

/*     b = solution vector, x . */


/*     forward elimination */

    /* Parameter adjustments */
    --ipvt;
    --b;
    a_dim1 = *ndim;
    a_offset = a_dim1 + 1;
    a -= a_offset;

    /* Function Body */
    if (*n == 1) {
      goto L50;
    }
    nm1 = *n - 1;
    i_1 = nm1;
    for (k = 1; k <= i_1; ++k) {
      kp1 = k + 1;
      m = ipvt[k];
      t = b[m];
      b[m] = b[k];
      b[k] = t;
```

```
 i_2 = *n;
      for (i = kp1; i <= i_2; ++i) {
           b[i] += a[i + k * a_dim1] * t;
/* L10: */
      }
/* L20: */
    }

/*     back substitution */

    i_1 = nm1;
    for (kb = 1; kb <= i_1; ++kb) {
      km1 = *n - kb;
      k = km1 + 1;
      b[k] /= a[k + k * a_dim1];
      t = -b[k];
      i_2 = km1;
      for (i = 1; i <= i_2; ++i) {
           b[i] += a[i + k * a_dim1] * t;
/* L30: */
      }
/* L40: */
    }
L50:
    b[1] /= a[a_dim1 + 1];
    return 0;
} /* solve_ */

#ifdef uNdEfInEd
comments from the converter:  (stderr from f2c)
   solve:
#endif
```

frac

```
/* The following values are used to test for too small
   and too large values of v for an integer fraction to
   represent.

   These values are currently set for the Amdal 5890.
*/

#define MAX 4.0e+10
#define MIN 3.0e-10

double
```

```
frac(v, n, d, error)
double v, error;
int *n, *d;
{
      /*
        given a number, v, this function outputs two integers,
        d and n, such that

            v = n / d

        to accuracy
            epsilon = | (v - n/d) / v | <= error

        input:  v = decimal number you want replaced by fraction.
              error = accuracy to which the fraction should
                     represent the number v.

        output:  n = numerator of representing fraction.
               d = denominator of representing fraction.

        return value:  -1.0 if (v < MIN || v > MAX || error < 0.0)
                    | (v - n/d) / v | otherwise.

        Note:  This program only works for positive numbers, v.

        reference:  Jerome Spanier and Keith B. Oldham, "An Atlas
            of Functions," Springer-Verlag, 1987, pp. 665-7.
      */

      int D, N, t;
      double epsilon, r, m, fabs();


      if (v < MIN || v > MAX || error < 0.0)
            return(-1.0);
      *d = D = 1;
      *n = (int)v;
      N = (*n) + 1;
      goto three;

one:  if (r > 1.0)
            goto two;
      r = 1.0/r;
two:  N += (*n)*(int)r;
      D += (*d)*(int)r;
      (*n) += N;
```

```
  (*d) += D;
three:      r = 0.0;
      if (v*(*d) == (double)(*n))
            goto four;
      r = (N - v*D)/(v*(*d) - (*n));
      if (r > 1.0)
            goto four;
      t = N;
      N = (*n);
      *n = t;
      t = D;
      D = (*d);
      *d = t;
four: printf("%d/%d", *n, *d);
      epsilon = fabs(1.0 - (*n)/(v*(*d)));
      if (epsilon <= error)
            goto six;
      m = 1.0;
      do {
            m *= 10.0;
      } while (m*epsilon < 1.0);
      epsilon = 1.0/m * ((int)(0.5 + m*epsilon));
six:  printf("   epsilon = %e0, epsilon);
      if (epsilon <= error)
            return(epsilon);
      if (r != 0.0)
            goto one;
}
```

s115

The five following programs come also from NETLIB, and are extracts from the suite of vectorization tests **vector.f**, converted from FORTRAN to C by hand. The translation does not account for the offsets needed to convert FORTRAN 1-based indices into C 0-based indices, so the codes shown are not strictly equivalent to the originals, but show the same specific problems in a cleaner setting.

```
void
s115(double **aa, int n)
{
    int             i, j, k;

    for (j = 1; j < n; j++)             /* do 320 j */
        for (k = 1; k < j-1; k++)       /* do 320 k */
            for (i = k+1; i < n; i++)   /* do 320 i */
                aa[i][j] = aa[i][j] + aa[i][k] * aa[k][j];
}
```

s123

```
void
s123(double *a, double *b, double *c, int n)
{
    int             j = 0;
    int             i;

    for (i = 1; i < n; i++) {           /* do 50 i */
        j = j + 1;
        a[j] = b[i];
        if (c[i] > 0) {
            j = j + 1;
            a[j] = c[i];
        }
    }
}
```

s124

```
void
s124(double *a, double *b, double *c, int n)
{
    int             j = 0;
    int             i;

    for (i = 1; i < n; i++) {           /* do 60 i */

        if (b[i] > 0) {
            j = j + 1;
            a[j] = b[i];
        } else {
            j = j + 1;
            a[j] = c[i];
        }

    }
}
```

s131

```
void
s131(double *a, double *b, int n)
{
    int             m = 1;
    int              i;

    if (a[1] > 0)
        a[1] = b[1];

    for (i = 2; i < n-1; i++)               /* do 340 i */
        a[i] = a[i+m] + b[i];
}
```

s2710

```
void
s2710(double *a, double *b, double *c, double x, int n)
{
    int              i;

    for (i = 1; i < n; i++)                 /* do 790 i */

        if (a[i] > b[i]) {

            a[i] = a[i] - b[i];

            if (n > 10)
                c[i] = fabs(c[i]);
            else
                c[i] = 0.0;

        } else {

            b[i] = a[i];

            if (x > 0)
                c[i] = a[i];
            else
                c[i] = -c[i];
        }
}
```

pat.f

This and the next program are from [SmA88]. The FORTRAN version is described in the paper of the reference.

```
c     From [. PAT icpp 1988 {, page 62}.]

      real a(20), b(20), sum, max
      integer i, j
      call init

      sum = 0
      j = 20
      do 100 i = 1,20
         sum = sum + a(i)
         a(i) = b(j) + a(i - 1)          ! this seems like an error for i=1
         if (b(i).gt.max) max = b(i)     ! what does this do?
         j = j - 1
100   continue

      print 105, sum, max
105   format('sum : ', f8.5, ' max : ', f8.5)
      end
```

pat.c

This is pretty much a direct translation (again, disregarding the difference in index origin) of the `pat.f` program. The most important difference is that the original contains an illegal subscript: when `i` is 1, the original makes a reference to `a(0)`, which is presumably illegal FORTRAN. The C version adds a conditional, which introduces a little more structure for the parallelizer to work on.

```
/* Adapted from Smith and Appelbe, ICCP 1988,
 * ``PAT -- An Interactive Fortran Parallelizing Assistant Tool''
 */

int
main()
{
    float           a[20], b[20], sum, max;
    int             i, j;
    extern void     init();

    init();

    sum = 0;
    j   = 20;

    for (i = 0; i < 20; i++) {
        sum = sum + a[i];
        if (i > 0)
            a[i] = b[j] + a[i - 1];
        if (b[i] > max)
            max = b[i];
        j = j - 1;
    }
    printf("sum : %f8.5 max : %f8.50, sum, max);
}
```

steele

This program was contributed (in its FORTRAN form) by Dr. Guy Steele, from Thinking Machines Corporation (personal communication, 12 December 1988). It shows a variety of loci for parallelization.

```
/*
 * Adapted from Guy Steele, personal communication, Dec. 1988
 */

#include <stdio.h>
#include <math.h>
#define SQR(x)  ((x)*(x))

void
migrt(p0, p, v, nt, nx, nz, dt, dx, pp, itinc)
    int             nt, nx, nz;
    double          dt, dx;
    int             itinc;
    double          p0[nt][nx];
    double          p[nz][nx];
    double          v[nz][nx];
    double          pp[nz][nx][3];
{
    int             ix, iz, it, i;      /* loop indices */
    int             it0, it1, it2, it3; /* auxiliaries */
    double          ddx, ddz;           /* more auxiliaries */

    for (ix = 1; ix < nx; ix++)
        for (iz = 1; iz < nz; iz++) {
            v[iz][ix] = SQR(v[iz][ix]/2 * dt/dx);
            if (v[iz][ix] > 0.5){
                (void)fprintf(stderr, "ERROR: v*dt/dx = %g > 0.50,
                                 sqrt(v[iz][ix]));
                exit(1);
            }
        };

    for (i = 1; i < 3; i++)             /* parallelizable */
        for (ix = 1; ix < nx; ix++)
            for (iz = 1; iz < nz; iz++)
                pp[iz][ix][i] = 0.0;

    it1 = 1;
```

```
        it2 = 2;
        it3 = 3;

        for (it = 1; it < nt; it++) {
            it1 = it3;
            it0 = it1;
            it1 = it2;
            it2 = it3;
            for (ix = 1; ix < nx; ix++)
                pp[1][ix][it2] = p0[nt-it+1][ix];

            for (ix = 2; ix < nx - 1; ix++) /* parallelizable */
                for (iz = 2; iz < nz - 1; iz++) {
                    ddx = (pp[iz][ix+1][it2]
                           - 2 * pp[iz][ix][it2]
                           + pp[iz][ix-1][it2]);
                    ddz = (pp[iz+1][ix][it2]
                           - 2*pp[iz][ix][it2]
                           + pp[iz-1][ix][it2]);
                    pp[iz][ix][it3] = (2 * pp[iz][ix][it2]
                                       - pp[iz][ix][it1]
                                       + v[iz][ix] * (ddx + ddz));
                }
        };

        for (ix = 1; ix < nx; ix++)
            for (iz = 1; iz < nz; iz++)
                p[iz][ix] = pp[iz][ix][it3];
}
```

sploops

This program is another synthetic example that tries to exploit one of the loci of parallelization that appears in the previous program. The idea is to support what could be called coarse-grain vectorization, by which a serial loops gets converted into a series of parallel loops.

```
/* Coarse-grain vectorization */

void
mask_and_add(int n, int b[n])
{
    int             mask[n];
    int             i = 0;

    while (i < n) {
        /* form the mask */
        if (i % 2)
            mask[i] = 1;
        else
            mask[i] = 0;

        /* modify the target */
        if (mask[i] && i > 2)
            b[i] += 1;
        else
            b[i] = 0;

        i += 1;

    }
}
```

APPENDIX C

Graph Parsing Via The SSFG Grammar

Understanding the specifics of SSFG parsing is not needed to follow its use in this work, yet at least some notion is needed to understand the notation in the catalog. The reader is again referred to [FKZ76] for the proofs and every detail I omit here.

SSFG parsing begins by classifying every node in the given flow graph as either a *decision* (a node with out-degree 2) or a *computation* (a node with out-degree 1).

Then a set of rewrite rules is applied eagerly on this graph, meaning that, given that a rule matches locally, it can be applied immediately, regardless of other rules possibly matching there or the rule under consideration matching elsewhere. The rules apply whenever a subgraph of the original graph is isomorphic to the pattern of the rule. These isomorphisms are easy to handle by small, ad-hoc, matchers. One can understand the rules as the replacement of a small subset of the current graph by an even smaller subgraph.

Each rule converts the graph into another with fewer nodes or fewer arcs. Also, the rewriting rules form a Church-Rosser system. All of this guarantees that the repeated application of these rewrites eventually ends in a unique resulting graph.

If that ultimate graph contains exactly one node (which necessarily is then a computation node), the parse succeeds.

I show here the rules by means of illustrations. For each rule I show the pattern graph that triggers the reduction, and the reduced graph.

Assume that the original nodes are labelled somehow (so that each node is distinct). The illustrations that follow show how the new nodes of the reduced graphs gain labels derived from the originals, such that the new labels encode the reductions applied. Thus if the parsing finally succeeds, the label attached to the final node is actually a parse for the original program.

Seen from the top of Figure C.1, the rules are these:

cc Computation-computation elimination. This rule is essentially the *T2* transformation in the T1–T2 system [HeU72]. Its repeated application subsumes a linear segment of the program (a generalized basic block) into one node.

loop Self-loop elimination. This is the corresponding *T1* transformation

Actually, the original T1 and T2 transformations don't require that there be just one exit node. The next transformations catch the other cases that would have been reduced directly by either T1 or T2. However, by distinguishing these subcases, the SSFG imposes more structure on the flow graph. Thus, the parses are richer in parts than if we just reduced segments and loops and paid no attention to the conditional

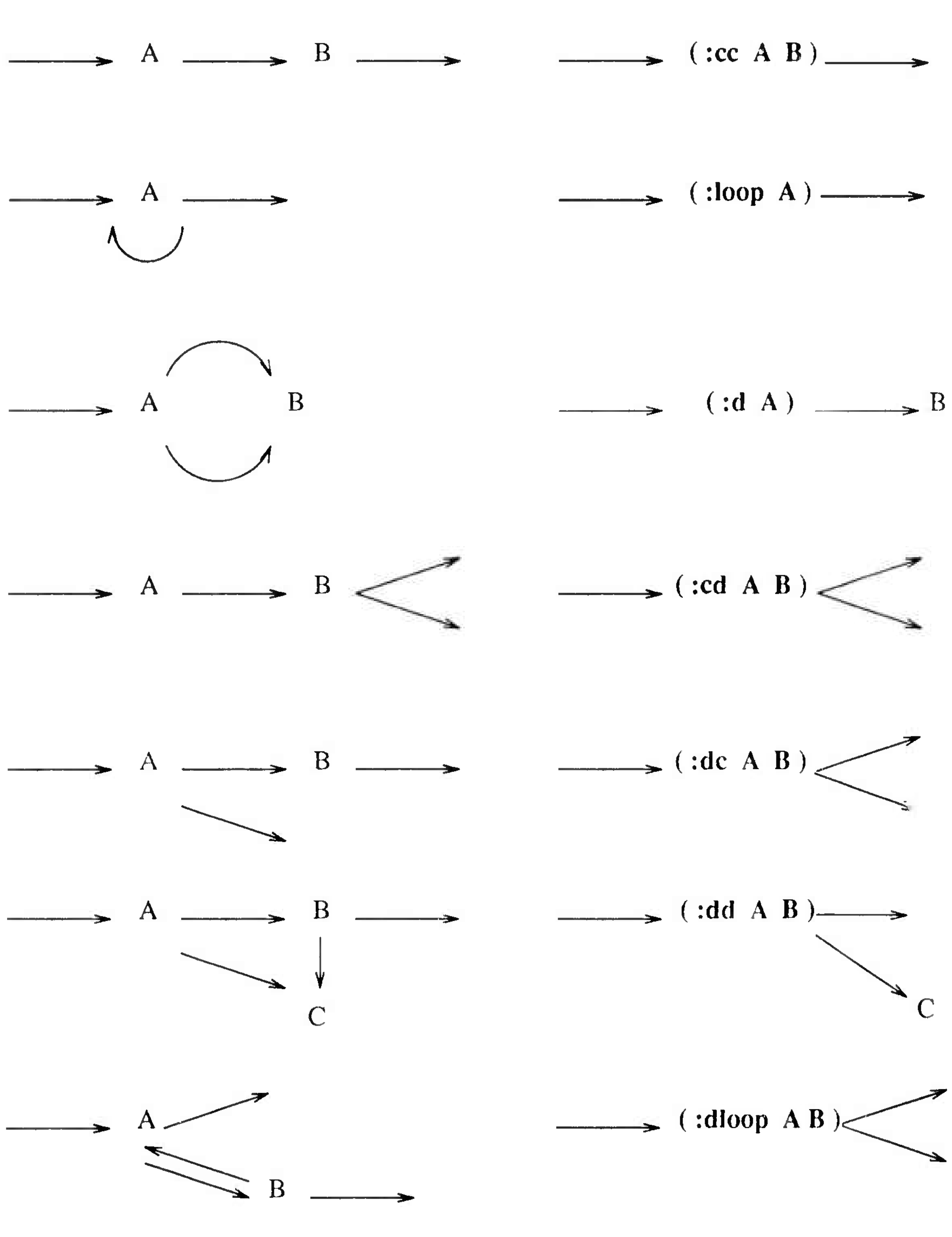


Figure C.1. SSFG Reduction Rules

branching structure.

d	Trivial decision elimination. If at a given step we have modified the graph so that a decision (at A) doesn't matter any more (flow goes to B anyway), we produce a computation node that remembers the lost decision.
cd	Computation followed by decision. This concentrates the calculations leading to a conditional.
dc	This is the complement of the above. Applied repeatedly on a branch of a conditional, it collects the entire branch into a node.
dd	Decision-decision elimination. Both conditionals yield C on occasion, so we reduce the graph to remember the compound conditional that selects C versus the rest.
dloop	This helps identifying cases of loops with more than one exit. Eventually the body will get distributed between two decision nodes, but the reduction would get stuck if this transformation were not performed.